NoSQL

[美] Joe Celko 著
王春生 范东来 译

Joe Celko's Complete Guide to NoSQL
What Every SQL Professional Needs to Know about Nonrelational Databases

权威指南

中国工信出版集团
人民邮电出版社
POSTS & TELECOM PRESS

NoSQL

Joe Celko's Complete Guide to NoSQL

What Every SQL Professional Needs to Know about Nonrelational Databases

权威指南

[美] Joe Celko 著

王春生 范东来 译

人 民 邮 电 出 版 社

北 京

图书在版编目（CIP）数据

NoSQL权威指南 /（美）乔·塞科（Joe Celko）著；王春生，范东来译. -- 北京：人民邮电出版社，2016.7
书名原文：Joe Celko's Complete Guide to NoSQL: What Every SQL Professional Needs to Know about Non-Relational Databases
ISBN 978-7-115-42787-8

Ⅰ. ①N… Ⅱ. ①乔… ②王… ③范… Ⅲ. ①数据库系统－指南 Ⅳ. ①TP311.138-62

中国版本图书馆CIP数据核字(2016)第134766号

内 容 提 要

本书是根据作者进行培训和开发的经验编写的 NoSQL 权威指南，是了解在什么场景、什么时候、为什么 NoSQL 的好处超过 SQL 的理想书籍。通过本书，读者可以对 SQL 的缺点多于好处的场景有一个完整的理解，而后更好地确定何时使用 NoSQL 技术可以获得最大的好处，对列式数据库、流式数据库和图数据库的利弊有更深入的理解，并在备受欢迎的 SQL 专家 Joe Celko 的指导下顺利过渡到 NoSQL。

本书会详细介绍 NoSQL 常见的数据库的历史、技术原理及其优缺点，这些数据库包括列式数据库、图数据库、流式数据库、键值数据库、文本数据库、地理信息数据库、指纹数据库、分析型数据库、多值数据库以及层次数据库等。

本书适合所有对 NoSQL 数据库感兴趣的技术人员阅读。

◆ 著　[美] Joe Celko
译　王春生　范东来
责任编辑　杨海玲
责任印制　焦志炜

◆ 人民邮电出版社出版发行　北京市丰台区成寿寺路 11 号
邮编　100164　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京天宇星印刷厂印刷

◆ 开本：800×1000　1/16
印张：12.5　2016 年 7 月第 1 版
字数：242 千字　2016 年 7 月北京第 1 次印刷
著作权合同登记号　图字：01-2014-8591 号

定价：45.00 元

读者服务热线：(010)81055410　印装质量热线：(010)81055316
反盗版热线：(010)81055315

版权声明

对本书的赞誉

“如果你遇到了通过 SQL 模型无法解决的问题，或者你想学习数据管理方面的知识，尤其是 NoSQL 方面，那么 Joe Celko 的书会是你理想的选择。”

——Jeff Garbus，飞鹰咨询合伙人

译 者 序

如果说某种技术是以“No”开头的，那么这种技术想必是会很有趣的。NoSQL 正好属于此类。当一个个有趣的名字（Cassandra、Dynamo、MongoDB 等）从我们眼前掠过时，我们才发现，“No”并不是想要否定什么，而是给了我们更多的选择。

1970 年，Codd 博士提出了关系模型，并给出了“Codd 十二定律”，自此关系数据库横空出世，逐渐占领了主流市场。得益于成熟的 SQL 标准，直到今日关系数据库仍然长盛不衰。SQL 也成为数据库中的必修课。关系模型也逐渐超越了层次模型和网络模型。

从 2003 年开始，Google 公司的 3 篇论文极大地推动了云计算、大数据技术的发展，而这些技术的发展也反过来刺激了大数据应用的落地。人们逐渐发现关系数据库在某些场景下越来越力不从心，高可用的需求越来越强烈，有些时候性能比一致性更加重要，不是每个查询都需要做连接……就在这时 NoSQL 如一股春风吹进数据库世界，但其实我们回头看看，Berkeley DB 早在 1991 年就已经问世，并且已经内置到了 BSD 里。如今，NoSQL 产品层出不穷，架构、实现原理都差异较大，并且其应对的业务场景也是大相径庭的。这给初学者带来了一些困难。那么，学习 NoSQL 的第一步就是了解它们。

如果你是 NoSQL 系统的支持者，那么你会很自然地想要去了解它，如果你是 NoSQL 系统的反对者，那么你就必须了解它。谁能帮你完成这个任务呢？Joe Celko 无疑是很好的人选，作为数据库专家，他的《SQL 权威指南》在国内已经多次再版，是很多人的数据库启蒙教材，另外他还参与了 SQL-89 和 SQL-92 标准的制定，另著有一系列数据处理和数据库相关的书籍，是一位高产的作者。

本书中，除了常规的键值数据库，对于地理信息数据库、流式数据库、指纹数据库甚至分析型数据库都有涉猎。本书并不关注 NoSQL 数据库产品本身，但是它能帮你理清这些产品之间的关系与特性，使你在根据业务场景构建自己的架构时更加得心应手。

NoSQL 从诞生之日起其初衷就不是取代关系数据库，而是作为关系数据库的一种有益的补充，不要因为它改变了现有秩序而对它敬而远之。可以预见的是，NoSQL 系

统和关系数据库将共存下去。读完这本书，我们会发现，No 有时并不是“no”，还可以是“not only”。

范东来

2016 年 5 月于成都

译者简介

范东来　北京航空航天大学硕士，成都数联铭品（BBD）科技有限公司大数据技术部负责人，大数据平台架构师，著有《Hadoop 海量数据处理》，译有《解读 NoSQL》，极客学院布道师，研究方向为并行图挖掘、模式分类。

王春生　网名“平凡的香草”，典型的“完美主义+强迫症+现实主义”综合体，追求完美并苛刻。新浪第一个大规模数据库平台的设计者和开发者，先后担任过 DBA、架构师、技术总监等职位，对数据库、网络、系统等环节的应用和开发均有丰富经验，曾参与翻译《Puppet 实战手册》。微博@平凡的香草。

引　言

“没有什么会比引入新秩序更难，因为创新者必须要面对那些在旧环境中已经做得很好的对手，以及那些在新环境中做得很好的冷漠者。”

——*Niccolo Machiavelli*[①]

在过去的几十年，我已经通过 Elsevier/Morgan Kaufmann 出版社出版了一系列的书，这些书几乎全部是关于 SQL 和 RDBMS 的。而这本书对行业媒体中所谓的大数据、新 SQL 或 NoSQL（我们这些极客非常喜欢流行语）做了一些概述。第一个创造或挖掘了新名词的专栏作家或博主很可能会在维基百科上有对其（新名词）进行的阐述，甚至可能会写成一本书来讨论。

由于 SQL 模型是事实上的主导数据库模型，因此任何其他模型都会被认为是挑战者。但是，我们应该用什么流行词来表示这种新的模型呢？我们已经在 SQL 中存放数 PB 的数据很多年了，因此，“大数据”这个词看起来并不合适。而且，SQL 每五年会引入一个新的 ANSI/ISO 标准并发布，并不是一夜之间就从“旧 SQL”变为“新 SQL”的。“新 SQL”这个名词让我想起了新可口可乐（New Coke[②]），新可口可乐并没有激发用户的信心，也没有获得成功。

在当下的所有流行语中，我觉得“NoSQL”是最好的，我把它读作“N. O. SQL”（就是“not only SQL”的简写），而不是其常被读作的“no SQL”。“no SQL”意味着在过去 40 多年中，数据库技术做得非常不好，但事实并非如此！很多使用 SQL 的人，尤其是我，在处理 IT 工作时，越来越坚信俗语中说的“手里有锤子的人会把眼中的一切都认为是钉子[③]”，但做 IT 工作并不是用锤子盖房子。

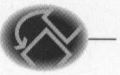
我们使用的一些数据库工具已经存在了几十年，甚至比 RDBMS 还要早。当然，由于技术发展使然，产生了很多新的工具。打开工具箱时，要考虑工具的所有可用选项，以及如何用它们完成任务。

---

① 尼科洛·马基雅弗利，意大利新兴资产阶级思想政治家，历史学家（http://en.wikipedia.org/wiki/Niccol%C3%B2_Machiavelli）。——译者注

② “kid with a hammer who thinks every problem is a nail”，参见 https://en.wikipedia.org/wiki/New_Coke。——译者注

③ 参见 http://en.wiktionary.org/wiki/if_all_you_have_is_a_hammer,_everything_looks_like_a_nail。——译者注



这本定位为概述的书可以带大家快速浏览一下你可能从没听说过或是已经忘记的老技术。然后，我们再研究“新玩意”，了解它为什么会存在。

我对硬件或深入研究某些特定软件不是那么感兴趣。这本书没有足够的篇幅讲这些，你可以针对自己感兴趣的或自己项目中某个聚焦的细节，阅读一些对应的书籍。请把这本书看成是百货公司的商品目录，通过它，你可以获得一些灵感，或是学到一些新的流行语。

请发送更正和意见至 jcelko212@earthlink.net，并在本书的配套网站（http://elsevierdirect.com/v2/companion.jsp?ISBN=9780124071926）中寻找反馈。

下面是你可能期待在这本书中能找到的内容的快速导览。

第 1 章：NoSQL 和事务处理。将作业队列读入大型计算机仍然是大量商业数据处理采用的方式。事务处理模型通过使用新的 ETL 工具来加载数据库，完成批处理作业。我们需要了解批处理和事务处理这两种模型，以及它们在新技术中如何使用。

第 2 章：列式数据库。列式数据库使用传统的结构化数据，并经常运行某些版本的 SQL，不同之处在于它们存储数据的方式。传统的面向行的方式被将数据以列的方式存放替代，并可将数据重新组装成人们在 RDBMS 模型中熟悉的行。由于这些列，每个列都被定义为一种且仅一种数据类型或域空间，因此，它们可以被压缩，并且可以分布于多个存储系统中，如 RAID。

第 3 章：图数据库。图数据库是基于图论的，图论是离散数学的一个分支。图数据库基于实体之间的关系建立模型，而不是根据这些属性的值做计算和聚合，以及根据这些值进行检索。

第 4 章：MapReduce 模型。在 IT 行业媒体上一般称为 NoSQL 或大数据的领域中，MapReduce 模型是最流行的。它用于在大型文件系统中并行地快速检索大量数据。这些系统在按照这样的速度和数量进行处理时，较少考虑数据完整性。基本操作是简单的，很少优化。但是，很多应用都愿意为此做出权衡。

第 5 章：流式数据库和复杂事件。关系模型和之前的传统数据库系统假定在查询时表是静态的，并且其结果也是一个静态表。但流式数据库是建立在不断流动的数据的模型之上的，可以认为是数据实时流动的河流或管道。流式数据库中最著名的例子是要在一秒内完成交易的股票软件和商品交易软件。这种系统必须根据这一流中的事件采取行动。

第 6 章：键值存储。从本质上来说，键值存储是键值对（<键>, <值>）的集合，可以理解为一类简单的数组。在每个集合内部，键都是唯一的。键可以是任意可以检验相等性的数据类型。这是 MapReduce 系列的一种形式，但其性能取决于键的设计方法。因此，散列成为很重要的技术。

模式与无模式。SQL 和所有以前的数据库都使用“模式”（schema）来定义它们的结构、

约束、默认值等。但是，使用和维护模式是有开销的。“无模式”的方式将所有的数据完整性（如果有的话!）放在应用程序中进行处理。但是，表现层却没有办法知道它们将被返回什么样的数据。这些系统是为检索优化的，SQL 系统的安全性和查询能力被更好的可扩展性和检索性能替代。

第 7 章：文本数据库。最重要的业务数据并不在数据库或文件中，而是在文本中。这些数据可能是合同/契约、担保条款、信件、手册和参考资料。文本天然就是模糊并且笨重的，传统的数据需要被精确和紧凑地编码。起初，文本数据库只能找到文件，但是随着算法的改进，我们已经能够理解阅读和文本的要点。

第 8 章：地图数据。地理信息系统（Geographic information system，GIS）是一个存储地理、地理空间或者时空数据的数据库。它不仅是一种地图制图。我们不仅是试图在地图上定位某样东西，还试图找出某个地区内这些东西的数量、密度和内容，在一段时间中的改变，等等。

第 9 章：大数据和云计算。大数据这个术语是由 Forrester Research 公司在白皮书中随着 4V 的流行语提出的，大数据的 4V 指的是：规模（volume）、速度（velocity）、多样性（variety）和可变性（variability）。大数据已经应用在目前我们已经讨论过并且试图协调混用多种数据库模型的场景下。

第 10 章：生物特征、指纹和专业数据库。生物特征还没有开始商用。它们用一个生物实体而非商业实体来识别一个人。我们处在一个医学和法制的世界。到最后，当安全性成为一个议题时，生物特征才可能会转到商用领域，并且我们希望安全地交易隐私信息。

第 11 章：分析型数据库。传统 SQL 数据库被用于联机事务处理。它的目标是为日常业务应用提供支持。联机分析处理数据库是构建在 OLTP 数据之上的，但是这种模型的目标是运行处理数据聚合的查询而非独立的事务。它是分析型而非事务型的。

第 12 章：多值数据库或 NFNF 数据库。RDBMS 是基于第一范式的，它假设数据以标量值的形式保存在行中的列里，这些记录有着相同的结构。多值模型允许用于表内嵌到列里。它们有一个不被 SQL 程序员所熟知的小众市场。这种数据模型有一种代数理论，听起来就像关系型模型。

第 13 章：层次数据库系统和网络数据库系统。IMS 和 IDMS 是最重要的前关系型技术，直到今天还在广泛使用。事实上，有一个好机会，IMS 仍然比 SQL 数据库保存有更多的商业数据。这些产品仍然在银行业、保险业和大型机上的大型商业应用里“繁忙工作”，它们使用 COBOL。它们对那种变化不大而又需要移动大量数据的场景非常适合。因为如此多的数据仍然存在于 IMS 和 IDBS 中，所以你必须至少了解分层数据库系统和网络数据库系统的基础知识，以便从中得到数据再用 NoSQL 工具保存。

作者简介

Joe Celko 为 ANSI/ISO SQL 标准委员会工作了 10 年，并且参与编写了 SQL-89 和 SQL-92 标准。

Celko 先生是 Elsevier/Morgan Kaufmann 出版的一系列关于 SQL 和 RDBMS 图书的作者。他是得克萨斯奥斯汀的一位独立咨询师。他已经为计算机行业和学术出版写过 1200 多个专栏，其中大多数都是关于处理数据和数据库方面的。

目　录

NoSQL 和事务处理

简介

本章讨论传统的批处理和事务处理。将作业队列读入大型计算机仍然是商业数据处理大量采用的方式。事务处理模型通过使用新的 ETL 工具来加载数据库，完成批处理作业。我们需要了解批处理和事务处理这两种模型以及它们在新技术中如何使用。

早期的时候，计算机系统只能做单路处理，也就是说计算机只能从头开始按照顺序完成一项作业。后来，有了多处理技术，多个作业可以共享计算机资源，但每个作业仍相互独立并在硬件队列中等着轮到自己执行。

这种方式演化为一种事务模型，并成为 SQL 数据库中使用的客户-服务器架构。事务处理系统的目标是要保证事务结束后特定的数据完整性，而 NoSQL 却不是这样的。

1.1 批处理中的数据库事务处理

让我们先从历史的角度看一下数据，以及它是如何变迁的。在有“大数据”（Big Data）之前就有“大型机”（Big Iron）的叫法，也就是，使用批处理方式工作的的大型计算机。每个程序都用自己独享的数据和处理器运行，与其他用户或资源不冲突。磁带或穿孔卡片一次只能读取一个作业。

批处理作业的调度是一个独立的外部操作。提交作业后，该作业会进入一个队列，由调度器决定它何时运行。系统告知操作员（是的，这是一项独立的工作！）挂载哪些磁带、把什么形式的文件加载到打印机等所有的物理细节。当作业结束时，调度器必须释放资源，以便

后续作业可以继续使用这些资源。

调度器必须保证队列中的每一项作业都可以完成。如果队列中的某个作业不断被赋予较高的优先级，可能就有其他作业无法完成。这就是所谓的活锁（live-lock[①]）问题。可以形象地理解为，小猪群中最小的那头小猪总是会被其他仔猪推离它的母亲，不允许它吃奶。针对这种问题，一种解决方案是，当作业在队列中等待 n 秒时，降低它的优先级编号，直到它最终到达第一的位置。

例如，如果两个作业 J1 和 J2 同时既要使用资源 A 又要使用资源 B，就会遇到死锁的情况。如果作业 J1 获取了资源 A 并等待资源 B，作业 J2 获取了资源 B 并等待资源 A，它们会等在那里一直等待下去，除非两个作业或其中一个作业释放其已经获取的资源，或者我们可以再找到这两个资源的另一个副本。

今天，大多数商业数据处理仍然是这样做的，只是磁带驱动器已换成了磁盘驱动器。

1.2　磁盘处理中的事务处理

磁盘驱动器发明的时候世界发生了改变。起初，磁盘驱动器被当作“快速磁带驱动器”，进行挂载和卸载，并被分配给单个作业。但数据库的特点是，它是有多个作业在同一时间运行的一个共同资源。

这种模型中没有队列。用户登录一个会话，会话连接到整个数据库。数据表并不是文件，用户也不会连接到一个特定的表中。SQL 引擎内部的数据控制语言（Data Control Language，DCL）决定哪些表可以被哪些用户访问。

如果批处理系统像一个豪华夜总会的看门人，决定谁可以进到夜总会里面，那么数据库系统就像一个处理满屋子各自做着自己的事情的表的服务员。

在这个世界上，与磁带一条记录一条记录地被读取相比，一个用户会话中可用的数据量是巨大的，可以有多个会话同一时间运行。处理流量是一个重大的概念变化和物理变化。

1.3　ACID

已故的 Jim Gray[②]在 20 世纪 70 年代才真正发明了现代事务处理，并在 1981 年 6 月写入经典论文“事务概念：优点和限制”（The Transaction Concept: Virtues and Limitations）。从这篇论文开始，有了 ACID（原子性、一致性、隔离性和持久性）这个缩写词。Gray 的论文论

① 参见 http://en.wikipedia.org/wiki/Deadlock#Livelock。——译者注

② 参见 http://en.wikipedia.org/wiki/JimGray(computer_scientist。——译者注

述了原子性、一致性、持久性，隔离性是后来补充的。Bruce Lindsay 和他的同事于 1979 年在 Gray 的论文的基础上写了论文“分布式数据库要点”（Notes on Distributed Databases），并制定了一致性的基本原理和数据库复制的首要标准。1983 年，Andreas Reute 和 Theo Härder 发表了论文“面向事务的数据库恢复的原则”（Principles of Transaction-Oriented Database Recovery），并创造了 ACID 这个名词。

ACID 这个术语的具体含义如下。

- *原子性*。事务中的任务（或所有任务）要么全部执行，要么全不执行。这是或全或无的原则。如果事务中的一个元素失败，则整个事务失败。SQL 坚守这一原则，`INSERT` 语句会将整个数据集插入表中，`DELETE` 语句会从表中删除整个一组行，`UPDATE` 语句会删除并插入整个数据集。
- *一致性*。事务必须在任何时候都满足系统定义的所有协议和规则。事务不能违反这些协议，同时数据库在事务开始和结束时必须保持在一致状态。在 SQL 中，这意味着在事务结束时所有的约束是 `TRUE`。这可能是由于系统的新状态是有效的，或者是由于该系统回滚到其初始的一致状态。
- *隔离性*。事务无法访问处于中间状态或未完成状态的任何其他事务的数据。因此，每一个事务自身都是独立的。在数据库中，性能和事务的一致性是必需的。但在 SQL 中不是这样，只有隔离级别的概念。会话可以在某些隔离级别看到未提交的数据。这个未提交的数据可以通过会话回滚，所以从某种意义上说，这些数据根本不存在。
- *持久性*。一旦事务完成，它将是完整、持续的，并且不能被撤销。即便是在系统故障、断电或是其他类型的系统宕机的情况下数据也能存活。这是一个硬件问题，但是我们仍然在这方面做了很好的工作。当然，我们只是不要让数据在易失性存储器中存储，而是尽快将其持久化。

这项功能（特性）在大多数 SQL 数据库中是通过各种“加锁”的方案来实现的。“锁”会决定其他会话如何使用资源，如只读取提交的行，或者允许读未提交的行，等等。这就是所谓的悲观并发模型（pessimistic concurrency）。其基本假设是，你必须保护自己免受其他人的影响，并且冲突是常态。

另一种流行的并发模型称为*乐观并发*（optimistic concurrency）。如果你了解过数字影片行业，你就能理解这种模式。每个人都会得到数据的一个副本，然后按照他们希望的那样进行处理。在缩微胶片系统中，影片管理员制作电影文档（影片）的副本，并把它们分发出去。每位员工将修改他自己的副本，并把它（修改后的影片、文档）提交到文档纪录中心。

这种模型中假设：

- 查询比数据库变更常见得多，专为查询设计；
- 数据库变更过程中冲突极少发生，将其当作例外；
- 如果确实遇到冲突，相关的会话可以选择回滚，或者可以设置解决的规则。等待情况恢复正常，避免恐慌。

对于缩微胶片系统，大多数请求是读取信息，数据从来没有修改过。对内容进行修改的请求通常在时间上是分离的，所以它们不会产生冲突。当一名或多名员工做了同样的修改时，并不会产生冲突，修改会直接生效。当两个员工的修改有冲突时，影片经理会两个修改都拒绝，然后他等待通过应用规则或是稍后再做的新修改。

乐观并发取决于每一行的时间戳，保持历史副本。用户可以在一个自己知道出局苦处于某个 ACID 状态的时间点来查询数据库。用微缩胶片的比喻来说，就像中央记录员在等待他人返回其标记（修改）的副本时的处理方式一样。但是，这也意味着，我们在“时间=t_0”时开始与该数据库交互，并且如我们所希望的那样，在“时间=t_0, t_1, t_2,…, t_n”时，基于所述时间戳同样能看到对应的数据。由于“锁”的作用，插入、删除和更新不会影响查询。如果要对一个恒定的流入数据进行查询，如股票和商品交易，乐观并发非常有用。

关于乐观并发的更详细信息将在关于流式数据库的 5.1.1 节中讨论。这种方法是最适于处理不断变化的数据，但又必须保持数据完整性，并在某个时间点呈现的数据一致视图的数据库。

需要关注的是：数据的集中管理方式并没有改变！

1.4　悲观并发详解

悲观并发控制假定冲突是预料之中的情况，必须警惕。在关系数据库管理系统（relational database management system，RDBMS）中最流行的模型是基于加锁的。锁是一种允许一个用户会话对资源的访问同时保持或限制其他会话对同一资源的访问的装置。每个会话可以针对资源获得对应的锁，对资源进行修改，然后在数据库中提交（COMMIT）或回滚（ROLLBACK）相应的操作。COMMIT 语句将修改持久保存，ROLLBACK 语句将数据库恢复到会话之前的状态。如果修改遇到问题，系统也可以做一个 ROLLBACK 操作。这时，锁会被释放，其他会话可以访问对应的表或其他资源。

有各种各样的“加锁”，但 SQL 的基本模型中有如下几种事务之间可以互相影响的方式。

- P0（脏写）。事务 T1 修改一个数据项，此时另一个事务 T2 在事务 T1 执行 COMMIT 或 ROLLBACK 之前也在修改同一数据项。如果 T1 或 T2 执行 ROLLBACK，系统会不清楚正确的数据值应该是什么。脏写很不好，原因是这样会违反数据库的一致性。假

设在 x 和 y 之间有约束（如 $x=y$），并且如果 T1 和 T2 单独运行，它们会各自保持约束的一致性。但是，当两个事务以不同的顺序写入 x 和 y 时（这种情况只有允许脏写才会发生），约束就很容易被打破。

- P1（*脏读*）。事务 T1 修改了一行数据，然后事务 T2 在 T1 执行 COMMIT 之前读取该行数据。如果 T1 执行 ROLLBACK，T2 读到的行是从未被“提交”的，因此可以认为是根本不存在的。
- P2（*不可重复读*）。事务 T1 读取了一行数据，事务 T2 修改或删除了该行数据，并执行 COMMIT 操作。那么，如果 T1 尝试重读该行，则可能会收到修改后的值或者发现该行已被删除。
- P3（*幻读*）。事务 T1 读取一组满足某些搜索条件的行 N，然后事务 T2 执行了某些生成了满足事务 T1（使用的）所有搜索条件的一行或多行。那么，如果事务 T1 以同样的搜索条件重新读，将会得到不同的行集合。
- P4（*丢失更新*）。事务 T1 读取了一个数据项后，事务 T2 更新相关的数据项（可能基于先前读出的数据），然后事务 T1（根据它较早读取的值）来更新数据项，并执行 COMMIT，这时会发生异常的更新丢失。

这些现象并不总是坏事。如果数据库仅用于查询，或是在工作日期间不会进行任何修改，那么就不会发生这些问题。如果数据库系统不需要设法保护自己避免这些问题，那么它将运行得更快。在某些情况下对数据进行更改也是可以接受的。

想象一下，有一张表，表中存储着世界上所有的汽车。我想执行一个查询，找到红色跑车司机的平均年龄。这个查询需要花一定的时间来运行，在此期间，汽车会报废、买入或卖出，新车会被生产出来，等等。但是，我接受符合 P1～P3 这 3 个现象的情况，因为平均年龄从我开始查询到完成查询的时候不会改变太多。第二位小数的变化并不重要。

可以通过设置事务隔离级别防止这些现象发生，这就是系统使用锁的方法。原始的 ANSI 模型仅包括 P1、P2 和 P3。其他定义最早出现在由 Hal Berenson 和他的同事（1995 年）撰写的微软研究院技术报告 MSR-TR-95-51“ANSI SQL 隔离级别的批判”中。

1.4.1 隔离级别

在标准 SQL 中，用户可以在会话中设置事务的隔离级别。隔离级别可以避免刚才谈到的一些现象，并给数据库一些其他信息。下面是 SET TRANSACTION 语句的语法：

```
SET TRANSACTION <transaction mode list>
<transaction mode> ::= <isolation level> | <transaction access mode> |
<diagnostics size>
```

```
<diagnostics size> ::= DIAGNOSTICS SIZE <number of conditions>
<transaction access mode> ::= READ ONLY | READ WRITE
<isolation level> ::= ISOLATION LEVEL <level of isolation>
<level of isolation> ::=
 READ UNCOMMITTED
 | READ COMMITTED
 | REPEATABLE READ-
 | SERIALIZABLE
```

可选的<diagnostics size>子句告诉数据库设置给定大小的错误信息列表。这是标准SQL功能之一，因此在个别的产品中可能没有这个功能。其原因是，单条语句中可能存在多个错误，数据库引擎应该发现这些错误，并支持在宿主程序中通过GET DIAGNOSTICS语句在诊断区报告这些错误。

<transaction access mode>子句是自解释的。READONLY选项表示这是一个查询，让SQL引擎可以放松一些（不用考虑冲突等问题）。READ WRITE选项告知SQL引擎，数据行可能会被修改，它必须注意上面提到的3个现象。

在当前大多数SQL产品中都实现的重要子句是<isolation level>。事务的隔离级别定义了一个事务的操作允许受到并发事务影响的程度。事务的默认隔离级别是SERIALIZABLE，但用户可以在SET TRANSACTION语句中明确地设置隔离级别。

每个隔离级别都确保每个事务将完全执行或完全不执行，而且不会有更新操作会丢失。当SQL引擎检测到无法保证两个或两个以上并发事务的串行化时，或当它检测到发生了不可恢复的错误时，可以自行启动ROLLBACK语句。

来看一下表1-1。表1-1展示了隔离级别和3个现象。Yes代表该现象在对应的隔离级别下是可能发生的。

表1-1 隔离级别和3种现象

隔离级别	P1	P2	P3
SERIALIZABLE	No	No	No
REPEATABLE READ	No	No	Yes
READ COMMITTED	No	Yes	Yes
READ UNCOMMITTED	Yes	Yes	Yes

在表1-1中：

- SERIALIZABLE隔离级别是保证那些不得不并发执行的事务与它们在串行顺序执行情况下产生同样的结果。事务串行执行是指一个事务执行完成后才开始下一个事务执

行。访问数据库的用户就像是在排队等候获得对数据库的完整访问。

- REPEATABLE READ 隔离级别是保证在用户会话期间为用户维护数据库的相同镜像。
- READ COMMITTED 隔离级别允许当前会话中的事务能够读到会话运行期间其他事务已经提交的行。
- READ UNCOMMITTED 隔离级别允许当前会话中的事务能够读到会话运行期间其他事务创建但不一定提交的数据。

不管事务隔离级别是哪种，在执行语句、检查完整性约束、执行与引用约束相关的引用操作等隐含读取模式（schema）定义期间，现象 P1、P2 和 P3 都是不应该出现的。我们不希望模式自己针对不同用户发生变化。

1.4.2 私有的隔离级别

我们已经讨论了 ANSI/ISO 模型，但厂商往往会实现一些专有的隔离级别。我们需要知道它们的工作原理，以便在工作中使用这些产品。ANSI/ISO 在会话级别为整个模式设置隔离级别。专有模型可能会允许程序员用额外的语法分配表级锁。微软公司使用的语法有这样的提示列表：

```
SELECT.. FROM <base table> WITH (<hint list>)
```

该模型可以采用行级锁或表级锁。如果它们采用表级锁，可以得到与 ANSI/ISO 一致的特性。例如，WITH (HOLDLOCK) 相当于 SERIALIZABLE，但它仅适用于指定的表和视图，且只适用于由正在使用的语句定义的事务的运行期间。

用“读者”（reader）和“写者”（writer）的概念是解释各种模式的最简单的方法。读者和写者这两个名字无需解释。

在 Oracle 中，写者是彼此阻塞的，数据将保持被锁定状态，直到 COMMIT、ROLLBACK 或不保存数据停止会话为止。如果两个用户试图同时编辑同一数据，当第一个用户完成操作数据后，数据便被锁定。锁继续被保持，即使这个用户正在处理其他数据。

读者不会阻塞写者：读取数据库的用户不会在任何隔离级别阻止其他用户修改同一数据。但 DB2 和 Informix 有所不同。例如，在 Oracle 中，写者会阻塞写者，但在 DB2 和 Informix，写者在 UNCOMMITTED READ 以上的任何隔离级别都会禁止其他用户读同一数据。在较高的隔离级别，锁定数据直到保存或回滚被编辑的数据，可能会导致并发问题。如果你正处在一个编辑会话中，其他任何会话都不能读你已锁定在编辑的数据。

读者阻塞写者：在 DB2 和 Informix 中，在 UNCOMMITTED READ 以上的任何隔离级别读者都会禁止其他用户修改同一数据。应用程序在 DBMS 中打开游标，每次读取一行，一边遍

历结果集一边处理数据，只有在这样的应用程序中读者才能真正阻塞写者。在这种情况下，DB2和Informix开始获取锁，并在处理结果集的时候持有锁。

在PostgreSQL中，直到更改该行的第一个事务被提交给数据库或回滚的时候，行才会被更新。当两个用户试图同时编辑同一数据时，第一个用户会阻塞其他用户更新该行。直到该用户保存了修改，提交了对数据库的更改，或在不保存的情况下停止该编辑会话（回滚所有在该编辑会话中进行的编辑操作），其他用户才能编辑该行。如果使用PostgreSQL的多版本并发控制（MVCC）（这是默认和推荐的方式），写入数据库的事务操作不会阻塞读者查询该数据库。不论使用默认的READ COMMITTED隔离级别还是设置隔离级别为SERIALIZABLE，这都是成立的。读者不会阻塞写者：无论在数据库中设置哪个隔离级别，读者都不锁定数据。

1.5　CAP定理

2000年，Eric Brewer在ACM分布式计算原理主题研讨会做了主题演讲，并介绍了CAP定理（也称Brewer定理）。2002年，在麻省理工学院的Seth Gilbert和Nancy Lynch的努力下进行了修订和修改，后来又有很多人参与。

这个定理是针对分布式计算系统的，而传统并发模型会假设有中央并发管理机制。悲观并发模型有一个“交通警察”，乐观并发模型有一个“服务领班”。CAP代表一致性（consistency）、可用性（availability）和分区容错性（partition tolerance）。

- 一致性。与ACID是同样的概念。系统可靠地遵循其既定的数据内容规则吗？一个集群中的所有节点都能看到它们应该看到的所有的数据吗？不要简单地认为这是最基本的以至于没有数据库会违背它。有一些安全数据库会常常欺骗某些用户！例如，当我们登录到《星球日报》报社的数据库时，我们被告知，Clark Kent是一个温和的记者，为一家伟大的都市报纸工作。但如果你是Lois Lane，系统会告诉你，Clark Kent是一位超人，一位来自另一个星球的陌生访问者。
- 可用性。是指请求的时候数据库服务或系统是可用的。每个请求会获得失败或成功之外的反应吗？可以登录和连接会话到数据库吗？
- 分区容错性或健壮性。是指一个系统即使数据丢失或系统故障仍能继续运行。单个节点故障不应该导致整个系统崩溃。我在看我的3条腿的猫——它是分区容错的。但是，如果它是一匹马的话，我们可能就会射杀它（3条腿的猫是能够站稳的，但3条腿的马却不能）。

分布式系统只能保证CAP理论中的两个特性，而不是所有3个特性。如果需要可用性和分区容错性，可能不得不在一致性方面做出让步，并且忘记ACID。从本质上讲，系统说“我

能让你访问到一个节点，但是我不确定你找到的数据是否正确”或者“我可以保证可用性和数据是正确的，但不能保证它是完整的。”这就像关于软件项目的一个古老的笑话：你可以让项目按时交付、满足预算要求或是正确性的，但只能在其中选择两样。

为什么我们不想失去以前的优势？我们很想保持这些优势，但是大型计算机在大数据面前已经无能为力，并且“大数据”已经遍布各个领域。不再有中央计算机，今天每个企业都要在网络上处理数百、数千或者数万个数据源数和用户。

我们一直认为“大数据”在推动硬件限制的改进。在大型机的辉煌岁月中有一个古老的笑话：你需要做的就是购买更多的磁盘来解决任何问题。

如今，数据体积使用了以前从未用过的术语，SI 前缀 peta（PB，10^{15}）和 exa（EB，10^{18}）在 1975 年的第 15 届 Conférence Générale des Poids et Mesures（CGPM）被通过。

1.6 BASE

世界上现在满是巨大的分布式计算系统，如 Google 的 BigTable、Amazon 的 Dynamo 和 Facebook 的 Cassandra。这里我们要提到的 BASE 是下面内容的简写。

- 基本可用（basically available）。这意味着该系统按照 CAP 定理保证了数据的可用性。但可以响应“失败”、“不可靠的”，因为所请求的数据是处于不一致或是正在改变的状态。你玩过魔术八球[①]吗？
- 软状态（soft state）。系统的状态可能随着时间的推移而有所变化，所以即使在没有输入的情况下系统状态也可能会有变化，这是由于系统的目标是“最终一致性”，因此，与硬状态（这里指数据是被确认状态，也可以理解成数据状态时最终期望的状态）相对而言系统一直处在软状态。当然，系统的某部分可以存有最终数据，如像存储地理位置这样的常量表。
- 最终一致性（eventual consistency）。一旦系统停止接收输入，该系统最终将是一致的。这就给了我们一个可接受的短时间不一致的窗口。“可接受的短时间窗口”是一个比较含糊的说法，做非关键性的计算的数据仓库可以等待这个时间，但在线订单接收系统是务必要及时响应用户的，以保持客户满意度（短于 1 分钟）。还有另一个极端，实时控制系统必须立即做出响应。域名系统（DNS）是已知最常见的一种使用最终一致性的系统。域名的更新是通过协议和时间控制的缓存进行的，最终，所有的客户端将获得该更新。但更新周期已经远远超过了“瞬间”或是中心服务器更新的时间。这种需要一个全局时间戳，以便每个节点都能知道哪些数据项是最新版本。

① 参见 http://en.wikipedia.org/wiki/Magic_8-Ball。——译者注

与ACID模型相似，最终一致性模型也有很多变种。

- 因果一致性（causal consistency）。如果进程A已经给进程B发送了更新，随后进程B所做的后续访问都将返回更新后的值，并且新的写操作能够覆盖之前的内容。与进程A没有因果关系的进程C所做的访问会遵从正常的最终一致性规则。在早期的网络系统中这也称为伙伴系统（buddy system）。如果一个节点无法得到权威的数据源，它会问它的伙伴是否已经得到了新的数据病信任其数据。
- 读你所写一致性（read-your-writes consistency）。进程A，在它已更新某一数据项后，会一直访问更新后的值，而绝不会看到旧的值。这是因果一致性模型的一个特例。
- 会话一致性（session consistency）。这是前一个模型的一个实用版本，其中进程在会话中访问存储系统。只要会话存在，系统会保证“读你所写一致性”。如果会话因为发生故障终止，将会创建一个新会话并恢复之前的处理过程，并且保证不会重复操作。
- 单调读一致性（monotonic read consistency）。进程只返回最近的数据值，它永远不会返回任何之前的值。
- 单调写一致性（monotonic write consistency）。在这种情况下，系统保证数据通过同一个进程顺序地写入。不保证这个级别一致性的系统是很难开发的。可以把它认为是在网络中某个节点的本地队列。

上面这些特性是可以组合的。例如，单调读一致性结合会话级一致性就是其中一种组合。从应用角度来看，单调读一致性和读你所写一致性是最终一致性系统中最需要考虑的，但也并非总是必需的。这两个特性使开发人员可以更简单地构建应用程序，同时允许存储系统在一致性问题上更加宽松，并提供更高的可用性。

最终一致性往往采用同步或异步复制技术，已经成为RDBMS产品中备份系统的一部分。在同步模式下，复制更新是事务的一部分。在异步模式下，由于日志传输，更新可能会被延迟。如果数据库在日志传送前崩溃，备份数据可能是过时的或不一致的。基本上，不一致性的窗口取决于日志传送的频率。

1.7 服务器端一致性

在服务器端，我们可能在几个节点，但不一定是所有的节点，拥有相同的数据。如果所有 n 个节点都在一个数据值上达成一致，那么我们确定这个数据就是对的。生活是美好的。

但是，在针对“更新”建立共识的过程中，我们需要知道到目前为止邮件列表外有多少

节点确认收到更新。我们正在寻找一个仲裁规则来审计节点故障和不完整复制。这些规则与应用程序不同。大型银行转账可能想要在所有节点上完全一致。一个网站购物车应用程序只要满足下面的条件就是令人满意的：客户返回给任何服务节点购物车的某个版本，用户就可以继续购物，即使有一些丢失的物品也没关系。只要确保当用户点击“结账”按钮时其他节点知道要删除其购物车的本地副本就可以。

我们不希望将节点的突发紧急重新启动作为默认动作。早期的文件系统是就是以这种方式工作的。几十年前，我的妻子为处理社会保障数据的保险公司工作，一个打卡故障会中止整批处理，并发出无用的错误消息。

我们希望系统会设计有优雅的降级机制。Sabre 的航空订票系统会排出少量的重复订票。如果某个人有两次冲突的或冗余的订票，不会有什么问题，因为一名乘客无法同时做两个座位，在同一个座位也无法乘坐两次，问题会通过人为方式或者问题本身的逻辑来解决。

当某一个节点过载时，你可能会容忍降低性能并将部分负载迁移到其他节点，直到第一个系统得以修复。最好的例子是独立磁盘冗余阵列（redundant array of independent disks，RAID）系统。当一个磁盘发生故障时，它在物理上从阵列中删除并插入一个新的单元来取代它。在故障磁盘重建的过程中，访问的性能将会有一点点下降。在系统继续运行其日常任务时，数据必须从被替换阵列磁盘中复制到新磁盘。

1.8 错误处理

错误信息有两大类，我们可以遇到一些预料之中的问题，如无效密码，针对这些情况可以采用标准的响应或处理过程。假如我们忘记了正确的密码，并且在做多次尝试后仍不能使用正确的密码，就会被锁定。

第二类错误消息能告诉我们发生了什么事，可能会有使人厌烦的细节。这些信息会让用户进行一些处理操作或者让用户知道他为什么会失败。

但是有了 NoSQL 的发展和最终一致性模型的出现，事情也未必就会变得很舒服。系统还是会停止或锁定，不知道是为什么，可以做什么，或者需要多长时间来解决（如果能解决的话）。截至 2012 年，Twitter 用了一年多的时间试图从 MySQL 迁移到 Cassandra。有些用户希望对自己关注的人能即时反馈（如 Twitter 用户），任何延迟都太太长。2011 年 8 月，Foursquare 公司声称，因为 MongoDB 故障引发了 11 小时的停机。

1.9 为什么在这些场景下 SQL 不能发挥作用

总结一下为什么你可能会希望打破 SQL 以及传统的 RDBMS 模型。

- 你不必只在一台机器或一个网络中存放数据。
- 大多数情况下，你用的根本不是你的数据。
- 数据非常大，无法把它存放在一个地方。
- 时间以及空间是不协调的。
- 数据不一定是 SQL 善于处理的结构化数据。

我们将用接下来的几章来讨论数据的新特点，以及它们需要的特殊工具。

总结思考

想要了解新技术，需要理解旧技术的基础知识。

参考文献

Gray, J. (1981). *The Transaction Concept: Virtues and Limitations*. http://www.hpl.hp.com/techreports/ tandem/ TR-81.3.pdf. Cupertino CA.

Berenson, H., et al. (1995). Microsoft Research Technical Report MSR-TR-95-51: “A critique of ANSI SQL isolation levels”. Redmond, WA.

列式数据库

简介

从打孔卡和磁带的年代开始，文件就是物理设备上连续的字节，访问的方式是从文件开始（打开文件）到文件结束（文件结束的标志为 TRUE）。是的，存储可以在磁盘上被分割成数据页，并且各种数据页可以通过指针链连接，但这种模型仍然与前面提到的打孔卡、磁带是相同的。后来，文件被拆分成记录（record，更多物理连续的字节），记录又被拆分成字段（field，仍然是更多物理连续的字节）。

文件被一条记录一条记录地处理（读/取一条，然后下一条）或按照物理存储位置顺序地处理（从头到文件结尾，或遵循一个指针链前进/后退 *n* 条记录，等等）。在这种模型中没有任何并行。这里还有一个假设：文件中记录（行）和记录中字段是物理排序的。为了使对这些数据的访问有效，花了大量的时间和资源来对记录进行排序。在磁带上无法做到随机存取，打孔卡上也无法做到。

到了 RDBMS 和 SQL 时代，这种文件系统模型仍然占主导地位。Codd 博士也陷入了困境。首先，他必须在所有表中拥有一个主键（PRIMARY KEY），主键对应于顺序文件的排列次序；后来，他意识到键就是键，没有必要在 RDBMS 中让其中一个键成为特殊的字段。但是，SQL 已经纳入了已有的技术特性，而且早期的 SQL 引擎也是建立在既有文件系统上的，所以在这个环节上卡住了。

列式模式采用了一种完全不同的方法。但它是一种可以与 SQL 和关系模型很好地一起工作的模型。在 RDBMS 中，表（bable）是一个具有完全相同类型的行（row）的无序集合。一个行是相同类型的无序列（column）的集合，每列保存与既定列相关的标量值。可以通过列名称来访问对应的列，而不是通过存储中的物理位置，当然，也可以使用"SELECT*"等简单

方式来访问，以节省要输入。

对应的逻辑模型如下：表是行的集合，有且仅有一种结构；行是一组列的集合；列是取自一个且仅一个已知域的标量值。存储通常会在传统的文件系统中遵循这种模式，用文件表示表，用记录表示行，用字段表示列。但这都与物理存储无关。

在列式模型中，我们设计一个表，并且使用列式数据库特有的数据结构来存储每个列。行和表可以从这些行重新组合生成。看看平常的表的设计，可以看到为什么它们称为垂直存储模型，而不是水平存储模型。

2.1　列式数据库的历史

列式存储以及倒排或不按顺序存储文件的方式并不是最新提出的。TAXIR 是 1969 年为生物学建立的第一个列式数据库存储系统。加拿大统计局于 1976 年实现了 RAPID 系统，并将其用于加拿大人口和住房普查数据的处理和检索，以及其他与统计相关的一些应用。RAPID 被拿来与世界各地的其他统计机构共享，并在 20 世纪 80 年代被广泛使用。直到 20 世纪 90 年代，它一直被加拿大统计局使用。

多年来，Sybase IQ 是市面上唯一一个可以商用的列式 DBMS。然而，当 OLAP（online analytical processing，联机分析处理）越来越流行后，其他产品厂商看到，可以将某些用于多维数据集和汇总的技术应用到更通用的数据库上。

由于一个列只有一种简单的数据类型，可以做很大程度的压缩。如果需要重建行，必须打破单一数据集模型并且为行进行编号，就像在一个顺序文件中所做的一样。每一行是通过检查该行行号进行重构的。具有相同的数据值的连续行号可以按范围建模：`{start_position, end_position,data_value}`。这是最简单的压缩格式，但也是非常强大的。很多数据以离散值进行编码，因此在同一列中常常会有大量相同的值。

然后，我们可以针对特定领域和所用的数据类型，使用内置的专门例程对数据值进行压缩。如果有某个域占用与数据值相同或者比数据值更大的空间，这就是很愚蠢的做法，而成本也随之（数据列更大）而来（占用了更多的空间，同时也降低了效率）。使用短记号为较长值构建查找表是很容易的，参考`{domain_token, data_value}`，现在模型变为`{start_position, end_position, domain_token}`。

例如，美国的电话号码中区号使用的正则表达式是`[2-9][0-9][0-9]`，这意味着我们至多可以有 800 个区号。代替这 3 个 ASCII 字符，可以用 SMALLINT 或 BCD[①]来生成记号，

① 二进制编码的十进制（Binary-Coded Decimal）。——译者注

并得到一个小的查找表。这个例子并没有获得很大的节省，但就是这样的小数据逐渐增加，形成了 TB 级数据库。另一个更有力的例子是美国城市和城镇的名字，截至 2012 年，美国大约有超过 30 000 个城市和城镇。2 字节的整数可以存储 65 535 个记号。很少有名字只有两个字母的城市，其中还有不少名字是重复的，如美伊利诺斯州首府斯普林菲尔德是最常见的小镇的名字，这也是这个名字常常用在电视节目（如《辛普森一家》）中的原因。同样，一个 2 字节的整数大概可以表示 180 年内的日期[①]。

这也为我们提供了能够加入一定的完整性约束的独立环节。例如，666 和 777 是有效的区号，但截至 2012 年还被没有被分配使用。同样，在电影中看到的 555 是不存在的电话区号。拨号前的 Klondike 5[②]是用来保证（表示）的电话号码不能被呼叫的。在美国老电影、动画片以及现代美国电视节目中都回听到这个区号。我们可以在查找表中简单地将这些无效的值去掉！

各种形式的文本压缩方式是众所周知的。对于数据库的数据来讲，我们希望压缩是无损的，这也就意味着，当我们解压的时候，得到的内容与压缩前的内容是完全一致的。在音乐和视频中，为了速度和空间，在不伤及内容的情况下可以牺牲一些质量。Lempel-Ziv（LZ）压缩方式是无损存储中最流行的算法。DEFLATE[③]是针对解压缩速度和压缩率而优化的 LZ 的变体，但压缩速度可能会很慢。DEFLATE 用于 PKZIP、gzip 和 PNG 等。LZW（Lempel-Ziv-Welch）用于 GIF 图片。另外，值得注意的是 LZR（LZ-Renau）方法，它是 Zip 方法的基础。

LZ 方法使用重复字符串的替换表，表本身往往是霍夫曼编码的（如 SHRI、LZX）。这种方法非常适用于人类语言的压缩，因为语法词缀和结构词（介词、连词、冠词、小标记等）构成了大量的文本。但编码方案也倾向于遵循模式（pattern）。在美国，银行账号以美国银行家协会银行编号的代码开始的，产品序列号中会包括产品线等内容。分层和矢量编码方案采用了这种压缩方式。

近几年，已经用最小完美散列算法[④]做了大量的工作（如果还不太了解这一技术，请参见“散列快速入门”）。当列是一组相对静态的字符串时，这种方式是最理想的。任何值的集合，甚至那些没有共同子串的值，被散列成短的、固定长度的、能够被更快地进行计算的数据记号。

对这种模型比较幼稚的看法是，它是复杂的、大的、缓慢的。这只说对了一半。与简单的顺序字段读取相比，它的确是复杂的，但是它并不慢，而且也不大。

---

① 180 年大约是 65 700 天。——译者注

② 参见 http://en.wikipedia.org/wiki/555_(telephone_number)。——译者注

③ 参见 http://en.wikipedia.org/wiki/DEFLATE。——译者注

④ 参见 https://en.wikipedia.org/wiki/Perfect_hash_function#Minimal_perfect_hash_function。

有一个观点是计算机处理单元（CPU）是瓶颈，但随后的 SMP（对称多处理）、群集和 MPP（大规模并行处理）技术让我们获得了数百或数千倍的 CPU，可以比以往任何时候的运行速度都快。有点儿 IT 民俗意味的摩尔定律[①]（Moore's law）认为每 18 个月计算机处理速度就会提升一倍，成本会降为之前的一半。但与此同时，数据量也越来越大。10 年前，20 GB 就被认为是无法控制的了，现在小型互联网创业公司已经开始管理 TB 级的数据。

MPP 使用许多独立的 CPU 并行地运行同一个程序。MPP 与 SMP 类似，但在 SMP 中，所有 CPU 共享相同的内存，而在 MPP 系统中，每个 CPU 都有自己的内存。

需要权衡考虑的是，MPP 系统中更难编程，因为处理器必须相互通信并相互协调。另一方面，SMP 系统在所有 CPU 试图在同时访问相同的内存位置时会发生阻塞。然而，这些 CPU 常常会发生闲置。这是由于 CPU 的 L1、L2（尤其是 L2）缓存当前缺少快速支撑大吞吐量数据的能力。在基于行的数据模型中，我们依然需要在 CPU 和存储间移动整条记录。

打一个比方，如个人电子产品，如果你只想从专辑购买几首曲目，在 iTunes 中只购买这几首曲目会很便宜。当你最想要专辑中大多数曲目，购买整张专辑会更加省钱和省时间，尽管你只播放其中的一些曲目。而“查询”与“搜索并检索”是不一样的问题。一条查询要么需要某个列要么不需要，并且在做查询编译时就可以知道需要哪些列，不需要哪些列。得益于现代处理器的速度和大型存储设备，把数据组装成行的方式在速度上远远快于从转动的磁盘中读单个字节的速度。2012 年，IBM 已经能够将 DB2 中的表的大小减为原始表的 1/3 或更小。由于`{domain_token, data_value}`这样的数据就是全部要读的数据，这种空间的节省带来很多回报，现在可以把数据放在更快的存储器上，如固态硬盘（SSD）。

列式存储和基于行的存储（数据库）之间的显著不同的是，一个表中所有的列都不会存储在数据页中。这避免了存储在数据页上的大量的元数据。这些元数据因产品不同而各异，但是最常见的元数据是每个数据页中的行（记录）相对起始位置的位移以及可能一些其他关于数据页的元数据。在这些元数据中还有每一行中的每一列相较起始位置的位移，以及这一列的值是否是 null。这就是 SQL 数据管理系统获知在磁盘何处放置读写磁头并返回哪些数据的方式。

特别是，在早期的 SQL 产品中向行中加入一个 `VARCHAR(n)` 列时，它们将按 `CREATE TABLE` 语句中列的顺序分配。不过，这也意味着这些列（这里特指 `VARCHAR(n)` 的列）将为全部 *n* 个字符分配存储，即便实际存储的数据很短（例如，实际的数据长度小于 *n* 的情况）。我们曾经使用的方案是手动地将所有的 `VARCHAR(n)` 列放到 DDL 语句中的尾部。目前，DB2 和 Oracle 已经在产品内部实现——在输出时对列进行重新排列，并且对用户隐藏这些细节。

① 参见 https://en.wikipedia.org/wiki/Moore%27s_law。——译者注

列式数据库模型可以就地（或在适当的位置）压缩数据，或者对指向字符串列表的指针进行存储。例如，first_name 与查找表{1 = Aaron, 2 = Abe,3 = Albert, …, 9999 = Zebadiah, …}对应。但是现在可以做一个折中，我们仍然可以将 VARCHAR(n)存储为固定长度，以加快搜索，但却并不会受到太大影响，这是因为查找表很小，查找表中的每个值只出现一次，因此我们还是会有相对较小的存储量。

显然，随着在数据页上插入记录或删除记录，这些元数据映射必须被更新。删除是很容易的，被删除的行可以立即被打上标记并在列被读取时被忽略。插入的新数据可以被添加到列结构的末端。尽管这种方式运行起来没问题，但最好仍然是将相关的数据做成集群[①]，以保持数据规模很小，并使数据值更容易搜索。有实用程序来恢复（还原）存储——其他内存管理系统的垃圾收集程序的列式数据库版本。

列式数据库也可以有索引，但实际上大多数没有。事实上，列式存储本身就是索引。位移也必须通过索引来处理。

散列快速浏览

索引和指针链涉及本地数据的物理搜索、查找。给定一个需要搜索的键，可以从指针链中的一个节点遍历到另一个节点，直到找到与搜索值对应的节点，或发现这个节点不存在。

散列是以数学为基础的磁盘访问技术。先定义好搜索键，然后把它放入公式中，公式会返回一个数组或查找表的位置。该位置包含物理存储地址，可以直接去访问它通过一次磁盘访问来找到该数据。

对于量较大的数据，散列比索引快得多。随着数据量的增加树形结构的索引会变得更深。在树中查找数据需要遍历，直至最终找到一个叶子结点满足查询条件。这可能意味着数据量大的情况下需要更多的磁盘访问。

两个或多个不同的搜索键会产生相同的散列键的情况是可能的，这就是所谓的冲突或（更文雅地说）散列碰撞。这种情况下，搜索键可以使用另一个函数重新进行散列，第二个函数常常是与第一个散列函数同族的函数，但是与第一个函数相比，会在一些常量上有所变化。有证据证明，某些散列函数最多 5 次重新进行散列计算，就会产生独一无二的结果。

没有冲突的散列函数称为完美的散列函数。如果散列函数在数组中没有空插槽，那么它就是最小的。一个最小完美散列函数需要满足：搜索值的集合是固定的，以便散列函数可以

① 这里的集群应该理解为“分片”（sharding）。——译者注

作用于编译器和类似的数据集中的关键字。

大多数散列算法的基本工具如下。

- *数字化挑选*。给定一个搜索键，从数中提取一些数字，也许会对这些数字重新排列。如果能够做到随机分布，就是一项很好的技术。事实上，这种方法被百货公司用在订货窗口上，使用姓氏的第一个字母对某些字母进行分类（S 代表史密斯，J 代表琼斯、约翰逊等，但几乎没有人会用到 X、Y 和 Z）。但是，客户的电话号码的最后两位数字是随机地均匀分布。
- *除法*。配有一个素数（m）的 mod（<键>，m）函数应该是非常好的散列函数（Lum et al.，1971）。它会给出(0, (m − 1))区间内的结果。TOTAL 和 IMAGE/3000 数据库[①]中会内置一系列大素数用来分配散列表。
- *乘法*。这种技术是对一个键做平方，然后取出中间的数字。5 位数的键将产生一个 10 位数字的平方，使用中间 3 位会具有良好的效果。例如，$54\,321^2$=2 950 771 041，中间的数字是 077 作为散列。散列必须来自中间的数字，否则可能会得到太多的聚类（即重复的结果）。
- *折叠*。这种技术取出 *n* 个数字的连续子集，并简单地将它们进行相加。例如，给定一个 5 位数的键，可以将所有 5 个数字相加，就获得了在范围（0≤散列值≤45）的结果。如果使用双数[②]，那么得到的范围将是（0≤散列值≤207）。这是一个很弱的技术，所以往往不会使用，或是被用来代替某些算法并且一般与另一种技术结合使用。

冲突解决技术也是多种多样的，最常见的一种是使用桶（bucket）。一个桶代表可容纳多个值的散列表的位置。下面是两种基本的方法。

- 开放寻址。这种方法试图在散列表中找到仍处于打开状态的桶。最简单的方式是从冲突的位置开始，并且将此点作为起点对开放空间进行地址递增式的线性搜索。其他类似的技术使用了比地址递增方式更加复杂的函数。常用的函数是使用二次方程式和伪随机数生成器。
- 外部链。可以将新的值添加到一个链表中，这个链表可能是存储在主存中或部分存储在磁盘上。但是在正确选择主存表的大小的情况下，溢出到磁盘的数据可以控制在主存中散列表大小的 15%以下。这种方法是非常有效的。

① 惠普公司的数据库管理系统数据。——译者注

② 即将 5 位数从左向右每 2 位进行相加，如果最后剩余一位就直接相加，则 5 位数做双数可以拆分成 99 + 99 +9 = 207。——译者注

2.2 技术原理

由于在列存储中的所有值都是同一类型的，并来自同一个域，计算其中第 n 行的位置很容易。所有列都按相同的顺序，因为它们在原始行中，所以要组装第 i 行，可以转到相关的列存储的第 i 个位置并且将它们连接起来。在电话号码的例子中，转到 area_codes、phone_exchange 和 phone_nbr 列存储并且在每一列中并行查找第 i 条记录。

区号相对较小，所以它们最先返回，其次是交易所，最后是电话号码。当我第一次在 Sand（nee 引擎）数据库中看到这个时，是非常令人惊讶的。测试数据是一组超过 500 万行的加利福尼亚州洛杉矶市的公共数据，并慢慢地逐步增加数据以进行监测。其结果在测试机的屏幕上以列的方式展现，而不是行的方式。在结果集中，这些列也没有按从左到右顺序在测试机的屏幕上呈现。

2.3 查询优化

有些列式数据库使用基于行的优化器，抵消了列式存储很多的优势。它们在具体化“行”之前在查询执行时使用基于行的优化器进行优化处理（只组装查询的列，实际上是做选择和投影）。基于列的优化可以将选择和投影分为单独的操作，这是 MapReduce 算法的一个版本（这些算法稍后会加以解释）。

目标是在查找实际的数据值前，获取尽可能多的行数。如果你能并行地收集列会更好。很显然，因为列中的数据已经完成，映射将首先开始。但选择操作需要尽快执行。

请注意，我刚才提到来自一个域的列。大多数实际数据库中完成的联结是等值联结，这意味着在不同表中的列都是来自同一个域，并且匹配相同的值。特别是，PRIMARY KEY 及其引用 FOREIGN KEY 都必须在同一个域中。PRIMARY KEY 列包含表的唯一值，而 FOREIGN KEY 可能是一对多的。

我们可以在列描述符中添加表的名字，使之成为域描述符：{table_name, start_position, end_position, data_value}。该载体可以是相当紧凑的，一个模式很少有 200 多万张可以用简单的整数进行建模的表。这种结构使某些连接操作转换为单域结构扫描。索引可以定位域描述符中的每个表的开始，并行访问相关的表。

2.4 多用户和硬件

列式模型的一个优势是，如果两个或更多的用户想使用列的不同子集，它们不需要彼此锁定。这种设计变得如此简单，是因为有了 RAID（redundant array of independent disks，最初

是 redundant array of inexpensive disks）的磁盘存储方法，这种方法将多个磁盘驱动器结合到一个逻辑单元中。数据存储在几种称为级别的模式中，具有不同量的冗余。冗余的概念是，当一个驱动器发生故障时其他驱动器可以接管。当更换的磁盘驱动器放置在阵列中时，数据从阵列中的其他磁盘被复制，然后系统被恢复。下面是 RAID 的各种级别。

- RAID 0（不带奇偶校验或镜像第块级带化）没有（或零）冗余。它提供了改进的性能和额外的存储，但没有容错。这是讨论的起点。
- RAID 1（条带，无奇偶校验或镜像），数据相同地写入两个驱动器，从而产生一个镜像集；读请求是由包含被请求数据的两个驱动器中任意一个具有最小寻道时间和旋转延迟的驱动器进行响应。这也是 Tandem 不停机计算模型的模式。停止机器需要一个特殊的命令“Ambush”，这必须在同一个临界点捕获两个数据流，所以它们不会自动重新启动。
- RAID 10（镜像和条带化），数据以条带化的方式写入主盘，并且在第二块磁盘上进行镜像。一个典型的 RAID 10 配置由 4 个驱动器组成：两个做条带化和两个做镜像。RAID10 配置采取了 RAID1 和 RAID0 的最佳概念，并将两者结合起来。
- 在 RAID 2（带有专用汉明码奇偶校验的位级带化）所有磁盘主轴的旋转是同步的，并且数据被条带化，所以每两个连续的位（数据存储单位）都分布在不同的驱动器。汉明码奇偶校验通过相应的字节计算并将其存储在至少一个奇偶驱动器中。这一 RAID 级别理论在实践中不使用。
- 在 RAID 3（带有专用奇偶校验的字节级带化）所有磁盘主轴的旋转是同步的，并且数据被带化，所以每个字节按照顺序分布在不同的驱动器。奇偶校验是通过相应的字节计算并存储在专用的奇偶校验驱动器上。
- RAID 4（奇偶校验的块级专用条带化）相当于 RAID 5，知识奇偶校验数据被存储在一个驱动器上。在这样的安排中，文件可以在多个驱动器之间分布。每个驱动器独立运作，允许输入/输出（I/O）要求并行完成。对数据库而言，并行性是一个巨大的优势。每个会话可以访问一个不带锁的大量引用的表或读竞争（只读锁）的表。
- RAID 5、RAID 6 和其他存在的模式，这其中的许多都是市场营销多于技术。我们的目标是提供驱动器故障的容错，甚至 *n* 块磁盘驱动器故障或阵列损毁。这使得更大的 RAID 阵列具有实际价值，尤其是对高可用性系统。当然这对数据库人员来讲是非常好的，可以从并行查询中得到更多的好处。

2.5　执行一个 ALTER 语句

ALTER 语句能够改变模式（schema）中的结构。在列式模型中，ADD COLUMN 和 DROP

COLUMN 是很容易的，创建一个新的列结构或从一个物理存储中删除旧的列结构。在面向行的模型中，每个行必须被压缩或延伸，索引也需要被重组。

因为同样的空间问题，在传统的面向行的数据库中修改数据类型也是很难的。在现实世界中，大部分变更是增加一列的物理存储。数字变得更大，字符串变得更长；只有日期类型似乎是不会膨胀的数据值，自从有了 ISO-8601 标准，一个固定的范围是 0001-01-01～9999-12-31。

在列式模型中，修改容易得多。将位置数据复制到一个新的列描述符，并将旧的数据值强制转换为新的数据值。当有新的列结构加载时，删除旧的，并添加新的。没有任何查询需要修改，除非它们具有特定数据类型的谓词（例如，如果日期成为整数，则 foobar_date<= CURRENT_TIMESTAMP 不会解析）。

2.6 数据仓库和列式数据库

当只涉及少数列时，数据仓库可以将部分工作负载转移到列式数据库，以提高性能。多维数据库（Multidimensional database，MDB）或多维数据集（cube）是支持非常快速访问预先计算的汇总数据的不同的物理结构。当查询请求 MDB 大部分列的情况下，MDB 将执行相当不错。

这些 MDB 数据库的物理存储是一个规范化的维度模型，通过存储计算结果消除了连接（join）操作。然而，随着列的增加，MDB 会变得巨大，并且增长速度会比预期快。MDB 中的数据可以使用与列式数据库大致相同的方式进行压缩，所以从多维数据集中提取列的子集相对容易。

对于列式数据库，最佳工作负载是查询，不去访问所用表的所有列就会有更好的性能表现。在这种情况下，少就是多。所需行的字节数越小，其性能就越好。

总结思考

很多重要的工作负载是与列选择相关的，因此可以在这个模型中获得巨大收益。列式数据库在大数据量、大规模扫描以及 I/O 密集型查询方面有很好的表现。在提供性能优势的同时，它们还具有独特的压缩其数据的能力。

列式数据库已经存在了一段时间，并在自己的优势方面表现非常好。但是，它们在当前市场得到特别的飞跃有两个原因。第一个原因是硬件改善，尤其是 SSD，使得主存储和辅助存储的区别不那么明显。当主存储和辅助存储在性能上存在巨大差别时，在辅助存储内压缩和解压缩数据和在辅助存储外压缩和解压数据是开销巨大的。而在 SSD 中却没有什么差别。

第二个因素是更好的算法。不但专门的压缩很不错，而且还有专为列式数据存储设计的并行算法。

参考文献

Lum, V. Y., Yuen, P. S. T., & Dodd, M. (1971). Key to Address Transform Technique: A Fundamental Performance Study on Large Existing Formatted Files. *Communications of the ACM*, 228–239.

图数据库

简介

本章讨论图数据库，它是用来为“关系”而不是为传统结构化数据建模的数据库。图数据库与演示类的图形图像没有任何关系。正如 FORTRAN 是基于代数理论的，关系数据库是基于数据集合的，图数据库都是基于图论（离散数学的一个分支）的。我们已经用另一种方式把一个思维工具转为一个的编程工具！

图数据库不是网络数据库。图数据库是通过指针链表将记录连接起来的预定义的关系数据库，指针链可以实现数据库中记录的逐条的遍历。IBM 开发的信息管理系统（IMS）就是这样一种仍在广泛使用的工具，它是一种分层访问模型。数据库集成管理系统（Integrated Database Management System，IDMS）、TOTAL 等产品使用更复杂的指针结构（如单向链表、双向链表、交叉链表等）允许更一般的图而不仅仅是一个树。这些指针结构是“硬指向”的，以便它们可被遍历，实际的数据也是按照这种结构来存储的。

在图数据库中，不是要试图做算数（数据计算）或数据统计，而是希望看到数据之间的关系。数据库专家和博主（http://www.dbms2.com/，http://www.monash.com）Curt Monash 为这种分析创造了一个词：关系分析（relationship analytics）。

程序员们应该在高中学过代数，并且可能在高中或大学接触过朴素集合论[①]。你可以利用高中的代数知识通过 FORTRAN 以及其他过程时语言进行编程，只有数学专业才需要高级集合论，其中会涉及无限集合（不管你的老板怎样认为，计算机都不会有无限的存储）。但是，

① 参见 http://zh.wikipedia.org/wiki/朴素集合论。——译者注

如果没有朴素集合论的知识，就不能真正理解 RDBMS 和 SQL。

但是，似乎只有数学专业的学生才会学习整个一学期的图论，这实在是太糟糕了。朴素图论中包含简单的概念和许多任何人都可以理解的日常生活中的例子。哦，我提到过简单的事情成为 NP 完全问题[①]会充满惊喜吧？让我们尝试着来弥补一下在教育上的欠缺。

在计算机科学中，我们用“大 O”符号 $O(n)$来表达能以输入大小(n)运行算法需要消耗多少工作量。例如，如果我们有一个简单的每个单位时间处理一条记录的进程，时间复杂度 $O(n)$是线性的，每增加一条记录，需要的执行时间就多了一个单位时间。但有些算法的时间复杂度增加是更复杂的方式。例如，文件排序可以在 $O(n\log_2(n))$内完成。其他算法可能具有更高的复杂度，如多项式平方和立方。然后，我们讲解 NP 算法。这些算法通常会尝试所有可能涉及的组合，以找到一个可行的解决方案，因此，这些算法的复杂度中会有一个阶乘，即 $O(n!)$。

NP 复杂度在许多经典的图论问题中都有所体现。新加入图中的每个节点或每条边都可能会带来越来越多的组合。我们常常会发现，我们实际上会不顾实际理由地寻找接近最优的方案。

3.1　图论基础

一个图中有两样东西：边（或弧）和节点（或顶点）。边被画成连接节点的线，节点被绘制为点或圆。就是这样的，分为两部分！不要将自己搞糊涂了，二进制数只有两部分，我们用二进制数来构造了计算机。

3.1.1　节点

节点是抽象的。它们通常是（但不总是）RDBMS 系统模型中的实体。事实上，图论的一些强大之处是，一个节点可以建模一个子图。在该模型中，一个节点可能具有也可能不具有“内部的东西”（电子元气组件），它可以静止在那里（公交车站），或仅仅是处于过渡状态。节点并不是一个对象。对象在其内部会包含方法和本地数据。在复杂的图查询中，我们可能会查找一个未知的甚至不存在的节点。例如，有保加利亚烧烤摊的公共汽车站可能不存在。但在保加利亚附近，可能会存在一个有烧烤摊的公共汽车站，直到我们综合分析了许多线索（例如，有车友在保加利亚文化中心公交站下车，才可以了解该站附近多个街区的餐馆或保加利亚教堂等）我们才知道它。你可能会看到的其他的图的例子还有以下几个。

- 简略地图：节点是公交车站、乡镇等。
- 电路图：节点是电气元件。

① 参见 http://zh.wikipedia.org/wiki/NP 完全或者 http://en.wikipedia.org/wiki/NP-complete。——译者注

- 状态转换图：节点就是状态（是的，这种图可以用 SQL 建模）。

3.1.2 边

边或弧会连接节点。我们把它们绘制成线，它们可以有箭头也可以没有，如果有的话，箭头用来表示流动的方向。在简略地图中，边表示道路，它们可以在其上标识出距离或时间。在电路图中，边用来表示导线，导线具有电阻、电压等。同样，抽象的状态转换也是由边连接的，这些边作为合法转换路径的模型。

在某种程度上，图论中的边比节点更有趣。在数据的 RDBMS 模型中，通过 REFERENCES 语句我们可以在表之间建立并未明确指定的单向关系。在图数据库中，不同节点之间可以有多条不同类型的“边”，这些边可以表示我们知道的各种特征（例如，如果你是一个星战迷，“我是你的父亲，卢克”“是笔友”）以及那些我们从其他“源”建立的关系（如“订阅《华尔街日报》”“朋友的朋友的朋友”“水手儿子的儿子”，如果你是一个巴菲特的粉丝的话）。

在抽象的最高级别，“边”可以是有向的也可以是无向的。在地图上，可以表示单向街道；在状态转换图中，这是“以前状态-当前状态”这样的状态对，等等。我们往往喜欢无向图，因为数学是更容易的，并且常常需要相反的某种类型的关系（如在《卢克·天行者》和《达斯·维达》的例子中的“父子”）。

从字面上意义上看，“有色边”试图数据库的显示上被着色的线。一种颜色用于展示同一类边、经典的“朋友的朋友的朋友”或使用社交网络时的 Bacon(n)关系（我会对此做简短的解释），社交网络服务会向你发送“你还可能认识……”的消息，并让你向对方发送邀请以建立直接关系的时候。

加权边可以进行累加，作为度量。在地图的例子中，距离就是度量，累加的规则是将边的权进行相加；在 Bacon(n)函数中，加权边用来减小两个节点（人）之间的间隔（你可能会问：“那是谁？哦，我忘记他了！”）。

3.1.3 图的结构

坏消息是，由于图论是相当新的数学标准，还不到 500 年历史，仍有很多未解决的问题，不同的作者会使用不同的术语。让我介绍一些人们普遍同意的基本术语。

- 空图（null graph）是指一组节点，没有任何边。完全图是指每对节点之间都有边。这两种极端在图数据库中都很罕见。
- 通路（walk）是连接一组节点的不重复的“边”的序列[①]。

① 参见 https://zh.wikipedia.org/wiki/图论术语。——译者注

- 连通图（connected graph）是指任意两个节点都可以通过一个通路到达的一组节点。
- 路径（path）是每个节点只经过一次的通路。如果你有 n 个节点，在路径中就有 n–1 条边[①]。
- 环路（cycle）或回路（circuit）是指能够返回到其起点的路径。在 RDBMS 中，我们不喜欢循环引用，因为引用操作和数据检索的循环引用可能引发无限循环并挂起。哈密顿回路是一个包含图中的所有节点的回路。
- 树（tree）是没有回路的连通图。我有一本关于如何为树和层级逻辑用 SQL 进行建模的书（Celko，2012）。因为层次数据库，我们倾向于认为有向树这种从属关系开始于一个根节点，并向下延伸到叶子节点。但在图数据库中，树并不像组织结构图那么明显，查找树中的节点是一个复杂的问题。特别是，我们可以以一个节点为根，寻找最小生成树。这是边的一个子集，代表图中根节点到每个节点的最短路径。

很多时候，我们缺少符合答案的边。例如，我们可能会看到两个人都收到了在一个特定的餐馆门前的交通罚单，但这意味着什么呢？只有我们看到他们的信用卡账单，才能知道这两个人曾经在一起吃饭。

3.2　RDBMS 与图数据库

作为一个概括性的描述，图数据库关注关系，而 RDBMS 则更关注数据。RDBMS 对于复杂的图论分析存在一定的困难。路径长度均为 1 的情况下，图还是比较容易管理的，这只需要一个 3 列的表（node、edge、node）。通过做自连接，可以构造长度为 2 的路径，等等，这种方式称为广度优先搜索。如果你需要一个脑海中的图像，可以考虑不断扩大搜索半径。如果有回路，你可能会快速进入笛卡儿爆炸来获得更长的路径，然后迷失在无限循环中。此外，如果路径长度是很长的、可变的或事先不知道的，通过写 SQL 进行图分析是非常困难的。

3.3　凯文·贝肯问题的六度

游戏贝肯数（Six Degrees of Kevin Bacon）是由 3 个奥尔布赖特学院的学生（Craig Fass、Brian Turtle 和 Mike Ginelli）在 1994 年发明的。他们在看电视的电影节目时发现，电影 *Footloose* 后的节目是 *The Air Up There*，引发大家猜测电影业中的每个人被以某种方式与凯文·贝肯有联系。凯文·贝肯本人也被赋予了贝肯数 0。曾在某部电影与他有合作的一个人被赋予贝肯数 1，与和他有合作的人一起工作的人的贝肯数是 2，以此类推。目标是寻找最短路径。截至

① 参见 http://en.wikipedia.org/wiki/Path_(graph_theory)。——译者注

2011 年年中，Oracle of Bacon 报道的最高有限贝肯数为 8。

这终究成为了一种时尚，并诞生了“Oracle of Bacon”（http://oracleofbacon.org）这个网站，它允许在任何互联网电影数据库（www.imdb.com）中的两个演员之间做在线搜索。例如，杰克·尼科尔森在电影 *A Few Good Men* 中与凯文·贝肯合作，并且在电影 *Wolf* 中合与米歇尔·菲佛作。我为位于犹他州的德雷珀的 Cogito 公司写了一个白皮书，在白皮书中我写了一段 SQL 用来查询凯文·贝肯问题，作为对他们的图数据库的基准测试。我希望更详细地谈论一些细节。

3.3.1 通用图的邻接表模型

下面是通用图的典型邻接表模型，包含通过上下文理解的一种边。结构在一个表中，节点在另一个单独的表中，因为它们是完全独立的不同事物（即实体和关系）。SAG 卡号指的是演员工会的会员标识符，但在下面的例子中我会使用单个字母来表示它们。

```
CREATE TABLE Actors
(sag_card CHAR(9) NOT NULL PRIMARY KEY
 actor_name VARCHAR(30) NOT NULL);
CREATE TABLE MovieCasts
(begin_sag_card CHAR(9) NOT NULL
 REFERENCES Nodes (sag_card)
 ON UPDATE CASCADE
 ON DELETE CASCADE,
 end_sag_card CHAR(9) NOT NULL
 REFERENCES Nodes (sag_card)
 ON UPDATE CASCADE
 ON DELETE CASCADE,
 PRIMARY KEY (begin_sag_card, end_sag_card),
 CHECK (begin_sag_card <> end_sag_card));
```

我要寻找从凯文·贝肯（在样例数据中用's'代表 start）到其他一些演员的长度不大于 6 的路径。其实，我真正想要的是演员之间的路径中的最短路径。

SQL 的优势在于它是一种声明式、面向集合的语言。当你为路径指定规则时，你会获得集合中的所有路径。通常来讲，这是一件好事。然而，这也意味着你必须计算并且拒绝或接受所有可能的候选路径。这意味着，你必须关注处理它们所需要的时间总和增加如此快，是否超过了宇宙中的计算能力。如果有一些试探法去除死路搜索将会更好，但目前还没有。

我做了一个决定，后来发现这个决定很重要的，我添加了 0 长度的自我遍历边（即一个演员与他自己总是在同一部电影中）。我将用字母来代替演员的名字。仅有 5 位演员，分别叫 {'s', 'u', 'v', 'x', 'y'}：

```
INSERT INTO Movies - 15 edges
VALUES ('s', 's'), ('s', 'u'), ('s', 'x'),
 ('u', 'u'), ('u', 'v'), ('u', 'x'), ('v', 'v'), ('v', 'y'), ('x', 'u'),
 ('x', 'v'), ('x', 'x'), ('x', 'y'), ('y', 's'), ('y', 'v'), ('y', 'y');
```

我对这种做法不是很满意，因为在开始寻找答案之前我必须判断路径中边的最大数量。但是，这种方式却会很好地运行，因为我知道路径都不会超过图中节点的总数。让我们创建路径的查询语句：

```
CREATE TABLE Movies
(in_node CHAR(1) NOT NULL,
 out_node CHAR(1) NOT NULL)

INSERT INTO Movies
VALUES ('s', 's'), ('s', 'u'), ('s', 'x'),
 ('u', 'u'), ('u', 'v'), ('u', 'x'), ('v', 'v'),
 ('v', 'y'), ('x', 'u'), ('x', 'v'), ('x', 'x'),
 ('x', 'y'), ('y', 's'), ('y', 'v', ('y', 'y');
CREATE TABLE Paths
(step1 CHAR(2) NOT NULL,
 step2 CHAR(2) NOT NULL,
 step3 CHAR(2) NOT NULL,
 step4 CHAR(2) NOT NULL,
 step5 CHAR(2) NOT NULL,
 path_length INTEGER NOT NULL,
PRIMARY KEY (step1, step2, step3, step4, step5));
```

我们开始查询，并加载所有可能的长度不超过 5 或者长度更短的路径的表：

```
DELETE FROM Paths;
INSERT INTO Paths
SELECT DISTINCT M1.out_node AS s1, -- it is 's' in this example
 M2.out_node AS s2,
 M3.out_node AS s3,
 M4.out_node AS s4,
 M5.out_node AS s5,
 (CASE WHEN M1.out_node NOT IN (M2.out_node, M3.out_node, M4.out_node,
M5.out_node) THEN 1 ELSE 0 END
 + CASE WHEN M2.out_node NOT IN (M3.out_node, M4.out_node, M5.out_node)
THEN 1 ELSE 0 END
 + CASE WHEN M3.out_node NOT IN (M2.out_node, M4.out_node, M5.out_node)
THEN 1 ELSE 0 END
 + CASE WHEN M4.out_node NOT IN (M2.out_node, M3.out_node, M5.out_node)
THEN 1 ELSE 0 END
 + CASE WHEN M5.out_node NOT IN (
M2.out_node, M3.out_node, M4.out_node) THEN 1 ELSE 0 END)
 AS path_length
```

```
FROM Movies AS M1, Movies AS M2, Movies AS M3, Movies AS M4, Movies AS M5
WHERE M1.in_node = M2.out_node
AND M2.in_node = M3.out_node
AND M3.in_node = M4.out_node
AND M4.in_node = M5.out_node
AND 0 < (CASE WHEN M1.outnode NOT IN (M2.out_node, M3.out_node, M4.out
node, M5.out_node) THEN 1 ELSE 0 END
+ CASE WHEN M2.out_node NOT IN (M1.out_node, M3.out_node, M4.out_node,
M5.out_node) THEN 1 ELSE 0 END
+ CASE WHEN M3.out_node NOT IN (M1.out_node, M2.out_node, M4.out_node,
M5.out_node) THEN 1 ELSE 0 END
+ CASE WHEN M4.out_node NOT IN (M1.out_node, M2.out_node, M3.out_node,
M5.out_node) THEN 1 ELSE 0 END
+ CASE WHEN M5.out_node NOT IN (M1.out_node, M2.out_node, M3.out_node,
M4.out_node) THEN 1 ELSE 0 END);
SELECT * FROM Paths ORDER BY step1, step5, path_length;
```

Paths 表中的 step1 列表示路径开始的位置，表中的其他列分别是第二步、第三步、第四步等。最后一步列是旅程的终点。SELECT DISTINCT 是一个安全的方式，“大于 0”的判断是要清理长度为 0 的“起点到起点”的路径。即使按照我的标准，这仍是一个复杂的查询。

路径长度的计算有点儿困难，CASE 表达式着眼于路径中的每个节点。如果它在该行中是唯一的，被分配值是 1；如果在该行中不是唯一的，它被分配的值是 0。

在这个路径表中存在 306 行。但实际上，这些行中究竟有多少是相同的路径呢？SQL 表中的列必须是固定数量的，但路径却可以是不同的长度。这就是说，$(s, y, y, y, y) = (s, s, y, y, y) = (s, s, s, y, y) = (s, s, s, s, y)$。路径中不应该有环路，因此你需要对答案进行过滤。能够做这种处理的唯一环节是在 WHERE 子句中或者在 SQL 之外的过程式语言中。

坦率地说，我发现在过程式语言中进行过滤比在 SQL 中更简单。加载每一行到链表结构中，并使用递归代码来查找环路。如果在 SQL 中这样做，则需要一个谓词来处理所有大小为 1、2……甚至多达图中所有节点总数的所有可能的环路。

在图数据库内部，也会使用一个简单的（节点，边，节点）存储模型，但它们会另外增加指针以连接附近的节点或子图。我对凯文·贝肯数据库做了一个基准测试。第一个测试用例是要找到以凯文·贝肯为“世界的中心”的度数，然后第二个测试是为了发现任何两个演员之间的关系。我用了 2 674 732 行数据。忽略建立数据的时间，查询时间的简单贝肯数列于表 3-1 中。这个计时记录了原始时钟时间，测试运行在同一硬件上，并且在每次开始时都清空了缓存。这个 SQL 运行在 Microsoft SQL Server 上，但后来用 DB2 和 Oracle 也获得了类似的结果。

表 3-1　贝肯数的查询时间

贝肯数	SQL	Cogito
1	00:00:24	0.172ms
2	00:02:06	00:00:13
3	00:12:52	00:00:01
4	00:14:03	00:00:13
5	00:14:55	00:00:16
6	00:14:47	00:00:43

如果我使用 SQL 进行广义搜索（如改变被检索的演员的名字，通过演员之间的联系，或通过一个共同的影片，并加入导演的名字等方式），那么这些数字（检索速度）会变得更糟糕。例如，改变被检索演员的名字可能会使检索速度变慢高达 9 000 倍，大多数会消耗几个小时，而不是像前面那样，少于一分钟。

3.3.2　通用图的覆盖路径模型

如果试图将所有路径都存放在 RDBMS 的一个有向图的表中会如何？该表看起来像下面这样：

```
CREATE TABLE Paths
(path_nbr INTEGER NOT NULL,
 step_nbr INTEGER NOT NULL
 CHECK (path_nbr >= 0),
 node_id CHAR(1) NOT NULL,
 PRIMARY KEY (path_nbr, step_nbr));
```

每条路径分配一个 ID 号及步骤，步骤从 0（路径的开始）开始编号，到 k（最后一步）结束。使用简单的带有 6 个节点的图，所有的单边路径是：

```
1 0 A
1 1 B
2 0 B
2 1 F
3 0 C
3 1 D
4 0 B
4 1 D
5 0 D
5 1 E
```

现在，我们添加 2 条边的路径：

```
6 0 A
6 1 B
6 2 F
7 0 A
7 1 B
7 2 D
8 0 A
8 1 C
8 2 D
9 0 B
9 1 D
9 2 E
```

最后是 3 条边的路径：

```
10 0 A
10 1 B
10 2 D
10 3 E
11 0 A
11 1 B
11 2 D
11 3 E
```

这些行可使用一个通用表表达式（common table expression，CTE），或是通过 SQL/PSM 这样的过程式语言中的循环从单边的路径产生。显然，只有少量的更长的路径，但随着边数的增加，路径数也会随之增加。当你得到一个真实大小的图的时候，路径的行数会是巨大的。但是，这仍然很容易找到 2 个节点之间的路径，具体如下：

```
SELECT DISTINCT :in_start_node, :in_end_node,
 (P2.step_nbr- P1.step_nbr) AS distance
 FROM Paths AS P1, Paths AS P2
 WHERE P1.path_nbr=P2.path_nbr
 AND P1.step_nbr <= P2.step_nbr
 AND P1.node_id = :in_start_node
 AND P2.node_id = :in_end_node;
```

注意 SELECT DISTINCT 的使用，因为大多数路径将是一个或多个长路径的子路径。没有它，在这个简单的图中搜索从 A 到 D 的所有路径将返回：

```
7 0 A
7 1 B
7 2 D
8 0 A
8 1 C
```

```
8 2 D
10 0 A
10 1 B
10 2 D
11 0 A
11 1 B
11 2 D
```

当然，只有两条不同的路径，即（A, B, D）和（A, C, D）。但在一个有很多连接的真实图形中，表中数据有很大可能会大比例返回。

我们可以做些什么避免大小的问题吗？有，也没有。在该图中，大部分路径是多余的，可以移除。寻找一组覆盖原始图中所有路径的子路径，对于这个简单的图用手工的方式也是很容易做到的：

```
1 0 A
1 1 B
1 2 F

2 0 A
2 1 B
2 2 D
2 3 E

3 0 A
3 1 C
3 2 D
3 3 E
```

在通用图中查找最长路径的问题是著名的 NP 完全问题，查找最长路径是找到最小覆盖路径集的第一步。对于一些非计算机专业的人来说，NP 完全问题会随着问题中元素的增多而明显地需要更多的资源（存储空间、时间）。在这些问题中通常都有穷举搜索和组合爆发的问题。

在这个模型中，尽管搜索查询是非常简单的，丢弃、修改或添加一条新边会修改整个结构，我们不得不重建整个表。组合爆发问题再一次出现，因此载入和验证表会消耗非常长的时间，即便仅仅是一个中等节点的数量。而另一个例子，MyFamily.com（Ancestry.com 的所有者）希望能够让访问者可以找到他们自己同一些著名人物之间的关系。这包括在一个超过 2 亿个节点以及 10 亿条边的图上，寻找长度为 10～20 条边的路径。查询的速率大概为每秒 20 次，即每天 200 万次。

3.3.3　真实数据的复杂关系

现在考虑另一种数据。假设你是一位负责犯罪现场调查分析的警察，手头只有各种各样

零散的事实，这些事实不能够给大家有效的、完整的关系表。这些事实以不同的方式将各种数据元素联系在一起。现在，你有 60 分钟时间通过各种目前还未知的方式来找出与坏人以及犯罪事实有联系的关联网络。

理想的情况下，你会做一个已知关系的“坏人”表和“可疑活动”表之间的连接。在编码前，你必须首先知道这样的连接组合是可能的。在收集到数据时将这些数据插到这些表中。你不能随意再建立另一种关系。

让我们来看一个实际的例子。警方以笔录和警察报告的形式收集监控数据，没有固定数据结构适应这个数据。例如，运输公司 U-Haul[①]汇报说，一辆卡车没有返回，然后他们就将其记录在警察报告中。就在同一周，一个农场供应公司报告，有人购买了大量硝酸铵化肥。在这两种情况下，如果是同一个人做了两件事情，并且用的是自己的真名（或是用已知的别名），你就可以根据“坏人”表将他们连接到一个关系里。这是相当容易的事情，你可以对每周的简单报告表进行这样的查询。这基本上是一个最短路径问题，意味着你正在努力寻找在美国的最愚蠢的恐怖分子。

在现实世界中，一个同谋者 A 租用卡车，另一个共谋者 B 购买化肥；或者某个人租了一辆卡车但不能及时返还，而另一个完全无关的人购买化肥使用了现金支付的方式，而不是使用转账到属于农民的帐户的方式。这种情况谁会知道呢？要找出这到底是巧合还是阴谋，需要知道参与人之间的关系。这种关系可以是弱关系（例如，两个人都住在纽约州），也可以是强关系（例如，他们在监狱是狱友）。

图 3-1 是该查询的截图以及回答这个查询的子图。看一下出租卡车丢失和肥料购买期间由样本数据生成的图。这是一个网络，是由两个刑满释放人员，加入卡车、化肥等因素的网络，他们具有相同的坐牢和参观水坝的时间。嘿，对任何人来讲，这都是一个重要标记！这种图形网络在统计和模糊逻辑学中称为因果图。如果你正在寻找数据的模式，你也会看到类似的方法，其称为鱼骨图（又称原因结果图或石川图）。在此之前，这种方法一直是“刮纸”技术。当你正在做针对一个非常引人注目的案子的 60 分钟警察秀时，有一个编剧，会非常棒！

在现实世界中，一个重要的警察部门每周有几百个案子。超级天才福尔摩斯这样的人物少之又少。即使你能找到这样的天才，在现实世界中你也根本没有足够的时间一案一例地通过白板做这种分析。如果希望按照这种方式运转（工作），在 21 世纪，情报必须是通过智能计算的。

大多数犯罪行为都是重复犯罪。惯犯倾向于遵循一定的模式，如果你关注连环杀手的话，会发现其中有些是相当可怕的。而警察部门要做的事情就是尽量去描述一个案件，然后通过

① 参见 https://en.wikipedia.org/wiki/U-Haul。——译者注

检查所有未结案的案来找出是否有两三个或更多案件具有相同的模式。

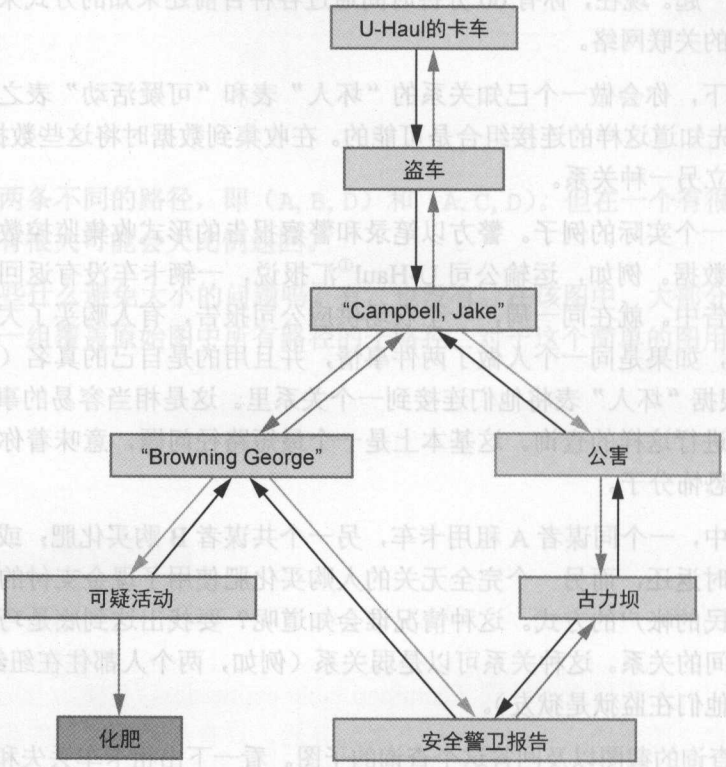


图 3-1　恐怖攻击猜想图

一个主要的优点是，数据直接进入图，而 SQL 则要求每个新“事实”必须针对现有的数据进行检查。然后 SQL 数据必须以某种比例进行编码来适应特定数据类型的列。

3.4　顶点覆盖

社会化网络营销依赖于找到“时尚的年轻人”——在一个社区中大家都知道的潮流达人。在某些情况下，他可能是一个人。例如，罗马教皇在天主教徒之间差不多是人所共知的，他的意见也很有分量。

形式上，一个无向图 G 的顶点覆盖是一个顶点集合 C，使得图 G 的每个边投射到 C 中至少一个顶点。集合 C 被说成覆盖 G 的边。非正式的一个例子，有一张奇怪的街道地图，想以这样一种方式在路口（节点）放安全摄像头，以使所有街道（边）都在摄像头的监视之下。我们还会谈到给节点着色，以将它们标记为一组期望的顶点的集合的成员。图 3-2 显示了两种

取自维基百科中有关此主题的文章中的顶点覆盖。

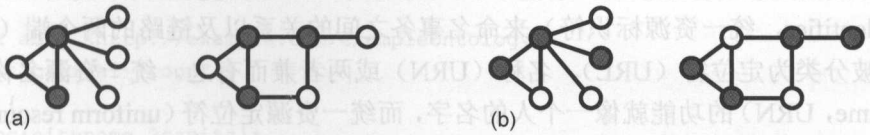


图 3-2 来自于维基百科的顶点覆盖：(a) 3 个节点的解决方案，(b) 4 个节点的解决方案

然而，这两种覆盖都是不小的。3 个节点的方案可以减少到 2 个节点，4 个节点的方案可以减少到 3 个节点，具体如下：

```
CREATE TABLE Grid
(v1 SMALLINT NOT NULL CHECK (v1>0),
 v2 SMALLINT NOT NULL CHECK (v2>0),
 PRIMARY KEY (v1, v2),
CHECK (v1<v2),
 color SMALLINT DEFAULT 0 NOT NULL CHECK (color>= 0));

INSERT INTO Grid (v1, v2)
VALUES (1, 2), (1, 4), (2, 3), (2, 5), (2, 6);
```

{1, 2, 6}和{2, 4}是顶点覆盖。第二个是最小覆盖。能证明这一点吗？在这个例子中，可以使用蛮力尝试所有可能的覆盖。找到一个顶点覆盖称为是一个 NP 完全问题，所以蛮力尝试是唯一可靠的方法。实际上，这并不是一个很好的解决方案，因为组合爆炸比你的思考得更快。

一种方法是估算覆盖的大小，然后随机挑选该大小的集合。然后保留最佳的候选节点，寻找共同子图，通过添加或删除节点修改候选节点。很显然，覆盖了边的 100%就赢了。这种方法可能不是最优的，但它是可行的。

另一个考虑是，该问题可能会开始于已知数量的节点。例如，你想给博主 n 个样品来宣传你的产品。你既希望给博主的礼物能覆盖最多读者，又要重复赠送最少。

3.5 图编程工具

截至 2012 年，并没有 ANSI 或 ISO 标准的图查询语言。我们必须依靠专有的语言或开源项目。它们依赖于底层的图数据库，这也是专有或者开源项目。商业提供商开始支持一些开源项目，Neo4j 是最流行的产品，并在开源世界中发展 10 年之后于 2009 年走到商用。这种事情在 PostgreSQL、MySQL 等关系数据库和其他产品上也发生过，所以不足为奇。

有一个针对资源描述框架（resource description framework，RDF）的ISO标准，这是在Web上进行数据交换的标准模型。它基于RDF扩展了Web的链接结构来使用URI（uniform resource identifier，统一资源标识符）来命名事务之间的关系以及链路的两个端（三元组）。URI可以被分类为定位符（URL）、名称（URN）或两者兼而有之。统一资源名称（uniform resource name，URN）的功能就像一个人的名字，而统一资源定位符（uniform resource locator，URL）类似于人的街道地址。换句话说，URN定义了一个元素的身份，而该URL提供了一种方法用于查找它。

这个差异很容易用一个例子解释。ISBN是一本特定版本的书的唯一标识。但要读一本书，你需要它的位置：一个URL地址。一个典型的URL是保存在本地硬盘上的电子书的文件路径。

由于Web是一个巨大的图数据库，很多图数据库建立在RDF标准上。这也使得拥有可以使用现有的工具的分布式图数据库更加容易。

3.5.1　图数据库

有些图数据库开始是在现有的数据存储系统上构建的，但随后被转移到定制存储引擎。这样做的原因很简单：性能。假设你要模拟一个简单的一对多关系，如凯文·贝肯问题。在关系数据库和SQL中，会有一个用来存储关系的表，表中包含对唯一实体表的引用，以及针对关系中的每一行数据进行多维度匹配的引用。随着关系表增长，执行聚合操作的时间增加了，因为你正在使用整个数据集。

在图模型中，在凯文·贝肯节点开始并遍历图，使用任何你想用的属性来查找边（如“与某人曾在一部电影中合作”）。如果节点有一个属性，根据这个属性进行筛选就可以了（如“这个演员是法国人”）。边表现得像非常小的、局部的关系表，但它们能提供遍历而不是连接。

图数据库可以有ACID事务。最简单的可能的图是一个节点。它可能是一条带有命名值（称为属性）的记录。从理论上讲，一个节点上属性的数量没有上限，但出于实用的目的，你会希望将数据分配到多个节点，按照明确的关系进行组织。

3.5.2　图数据库语言

在图数据库的世界没有等价的SQL。图论是数学的一个分支，所以基本术语和算法是众所周知的，延续着第一个产品。这是一个优势。但图数据库并不等于是IBM的System R——一个定义了SEQUEL（它最终成为了SQL）的研究项目。并不是任何人都试图制造一种语言进入ANSI、ISO或其他行业标准。

1．SPARQL

SPARQL（发音为“sparkle”，SPARQL协议和RDF查询语言的简写）是一种RDF格式

的查询语言。它试图利用常见的关键词而看起来有点儿像 SQL，又使用特殊的 ASCII 字符和 λ 演算而有点儿像 C。例如：

```
PREFIX abc: <http://example.com/exampleOntology#>
SELECT ?capital ?country
WHERE {
 ?x abc:cityname ?capital;
 abc:isCapitalOf ?y.
 ?y abc:countryname ?country;
 abc:isInContinent abc:Africa.}
```

其中?前缀表示一个空变量，:命名一个源。

2. SPASQL

SPASQL（发音为“spackle”）是对 SQL 标准的扩展，允许在 SQL 语句中执行 SPARQL 查询，通常被视为子查询或函数子句。这也允许通过“传统”的数据访问 API（ODBC、JDBC、OLE DB、ADO.NET 等）发出 SPARQL 查询。

3. Gremlin

Gremlin 是一种开源语言，基于取自面向对象和 C 编程语言家族中的语法，并根据属性进行图遍历（https://github.com/tinkerpop/gremlin/wiki）。具有针对有向边和更复杂查询的语法，看起来比 SQL 更数学化。下面是一个示例程序，顶点具有编号，从其中一个开始遍历。然后根据“likes”属性的输出路径来构建路径：

```
g = new Neo4jGraph('/tmp/neo4j')

// calculate basic collaborative filtering for vertex 1
m = [:]
g.v(1).out('likes').in('likes').out('likes').groupCount(m)
m.sort{-it.value}

// calculate the primary eigenvector (eigenvector centrality) of a graph
m = [:]; c = 0;
g.V.out.groupCount(m).loop(2){c++<1000}
m.sort{-it.value}
```

特征向量中心是一个节点在网络中的影响力的量度。连接高得分节点比连接到低得分节点有助于节点得分更高，在这个理念的基础上，它为网络中的所有节点给予相对分值。它可以在你的朋友名单中衡量“酷孩子”的效果。Google 的 PageRank 是特征向量中心测量的一个变种。

4. Cypher（NEO4j）

Cypher 是一个声明式图查询语言，仍在不断增长并越来越成熟，它会让 SQL 程序员感到

很舒适。它不是一个奇怪的 ASCII 字符的混合，其主要子句是人类可读的关键字。WHERE 和 ORDER BY 等大多数关键字是受 SQL 启发的。模式匹配借用 SPARQL 的表达方式。查询语言是由几个不同的子句构成。

- START：图中的起始点，通过查找索引或元素 ID 获得。
- MATCH：图模式匹配，在 START 子句中绑定到起始点。
- WHERE：过滤标准。
- RETURN：返回内容。
- CREATE：创建节点和关系。
- DELETE：删除节点、关系和属性。
- SET：属性设置值。
- FOREACH：对列表中的每个元素执行一次更新操作。
- WITH：将一个查询分为多个不同的部分。

例如，下面是一个查询，在索引中查找一个名字叫“John”的用户，然后找出 John 的朋友的朋友（但不是 John 的直接朋友），返回 John 以及通过遍历找出的朋友的朋友列表：

```
START john=node:node_auto_index(name = 'John')
MATCH john-[:friend]->()-[:friend]->fof
RETURN john, fof
```

我们从 john 节点开始遍历，MATCH 子句使用箭头来显示将“好友的好友”作为“边”构建到路径中。最后一条子句告诉这条查询返回的是什么。

在下一个例子中，我们根据一个用户列表（通过节点 ID），遍历图，寻找那些有向外扩展朋友关系的用户，只返回那些遵循名称属性以“S”开始的用户：

```
START user=node(5,4,1,2,3)
MATCH user-[:friend]->follower
WHERE follower.name=~ 'S.*'
RETURN user, follower.name
```

WHERE 子句与 SQL 及其他编程语言中的很相似。它有通常的逻辑运算符 AND、OR 和 NOT，比较运算符，简单的数学算式，正则表达式，等等。

5. 趋势

去 *http://www.graph-database.org/*阅读 PowerPoint 文件中提到的各种图语言项目。这将在未来几年不断变化，但你会看到几个趋势。建议的语言是声明式的，借用了 SQL 和 RDBMS

模型的思想。例如，GQL（Graph Query Language，图查询语言）的 SUBGRAPH 作为图数据库衍生表（子表）语法。很像 SQL，图查询语言必须将数据发送给外部用户，但它们缺乏一种标准的处理信息的方式。

获得一个声明式图查询语言的开源下载文件，并准备更新你的简历，这个目标非常值得努力。

总结思考

图数据库需要将思维模式从计算数据转变为计算关系。如果将要使用某种图数据库产品，你就必须学习数学书上有关图论的内容。好的入门书籍的简要清单列在下面的参考文献中。

参考文献

Celko, J. (2012). *Trees and hierarchies in SQL for smarties*. Burlington, MA: Morgan-Kaufmann. ISBN: 978-0123877338.

Chartrand, G. (1984). *Introductory graph theory*. Mineola, NY: Dover Publications. ISBN: 978-0486247755.

Chartrand,G., & Zhang, P. (2004). *A first course in graph theory*. New York: McGraw-Hill. ISBN: 978-0072948622.

Gould, R. (2004). *Graph theory*. Mineola, NY: Dover Publications. ISBN: 978-0486498065.

Trudeau, R. J. (1994). *Introduction to graph theory*. Mineola, NY: Dover Publications. ISBN: 978-0486678702.

Maier, D. (1983). *Theory of relational databases*. Rockville. MD: Computer Science Press. ISBN: 978-0914894421.

Wald, A. (1973). *Sequential analysis*. Mineola, NY: Dover Publications. ISBN: 978-0486615790.

MapReduce 模型

简介

本章讨论 Google 公司和雅虎公司为内部使用而开发的 MapReduce 数据处理模型。这是一种数据检索模型，而不是一种查询模型。

如我们所知，当今互联网，或你更喜欢的 Web 2.0，实际上是从使用任何人都可以用于构建网站的开源软件的 LAMP 系列技术栈开始的：Linux（操作系统）、Apache（HTTP 服务器）、MySQL（数据库，但因为它已经被 Oracle 收购，人们正在转向它的开源版本 MariaDB）和 PHP、Perl 或 Python 等应用程序语言。Apache 和 MySQL 现在由 Oracle 公司控制，它们的开源社区已经不再被信任。

在大数据存储领域也有一个类似的系列称为 SMAQ 存储系列：MapReduce 和查询，这不是指特定的产品本身。像 LAMP 系列一样，实现 SMAQ 各层的工具通常是开源的，运行在商用硬件上。这里最重要的词是“商用”，让更多的商店可以迁移到大数据模型。

这就引发了一个明显的问题是：什么是“大数据”。我发现最好的答案是：当无论因为任何原因使该项目相关的数据大小是关注的主要问题的时候，这个数据就是“大数据”。我们正在寻找被数据驱动，但不计算或分析数据的项目。第一个命中这个问题的 Web 应用程序是网络搜索引擎。这是有道理的，它们试图跟上网络的发展，而不丢下或放弃任何数据内容。

如今，有一些玩家在网络规模上遇到了难题。比较明显的是社交网络、多人游戏和仿真（模拟）项目，以及大型零售商和拍卖网站。但是，除 Web 服务以外，手机、传感器和其他常量数据流也可以产生 PB 级数据。

Google发明了MapReduce的基本技术，但在我撰写本书的时候，雅虎居然把它集成到了Hadoop存储工具上。基于Hadoop的系统有一个庞大的存储架构。因为Hadoop是用Java编写的，查询部分可以通过Java语言完成，但这些平台也有更高层次的查询语言（将来可能会更多）。

这个模型的核心部分是MapReduce。想象一下，一个大型开放式办公室中职员们坐在自己的办公桌（商用硬件）旁，在他们面前目录成堆。把目录放到自己的办公桌上是一个批处理流程，它不像SQL事务模型那样可以对数据库进行交互式地插入、更新和删除等操作。

继续办公室职员的这个例子，每天一次（或任何时间单位），邮件业务员会在职员办公桌上放置一天内所需要的目录（明细表）。职员们看不到的是，在邮件被放置在邮车上及分发之前，收发室（数据源）必须被清理、过滤，并经过一定的排序。大数据的数据仓库中的ETL（提取、转换、加载）工具也一样，但那些商业数据仓库使用的传统的结构化数据来源常常是不整洁的，这本身可以成为一个完整的话题。

但是，假设我们已经准备好开始工作。老板在办公室的前面向所有人喊：“喂，帮我找一双红色的芭蕾舞鞋！”在同一时间，一些职员可能会意识到，他们办公桌上的文件堆里没有“鞋”这个子目录，他们就会忽略这个请求。其他职员也会注意到他们捕捉到的老板的要求，并开始通过自己的目录查找。但是，他们使用什么样的逻辑来匹配老板的需求呢？大家应该知道，我们要求的是一个特定种类和颜色的女鞋。人类可以看图片并理解图片，但是计算机必须被编程才能做到这一点，并可能包括一个加权匹配分数，而不只是一个“是/否”的结果。算法越智能，需要的运行时间就越长，而且需要的资源也越多。

这是映射（map）部分。查询是必须并行执行的。在这个类比的例子中，喊出来的查询是足够的，但现实世界并不这么简单。你必须安排给每个人可以独立完成的任务并将所有结果合并成一个α结果。另一个邮件业务员需要接着运行，从命中（执行了查询）的职员的办公桌上收集结果。需要关注的是，职员们是以不同的速度完成的。一些职员没有与查询匹配的结果，可以跳过他们。有些职员在自己的目录中精确匹配了“红色芭蕾舞鞋”，一些职员有近似匹配，如“芭蕾舞鞋”或“红色舞鞋”。

现在是时候进入化简（reduce）阶段了。邮件业务员从职员那里获取了目录结果（记录笔记），送到房间前面的老板手中。但是，职员们的字迹潦草，所以邮件业务员要对结果进行总结，并且为这些笔记进行排序。现在需要更多的算法，并且需要关注表现层（结果展示）！最后，老板有他自己想要的答案。然后职员们开始准备执行另一个查询。

注意，这里更多的是“检索”，而不是SQL程序员认为的“查询”。细说下来的话，并不像关系型的划分、上卷、立体汇总或其他典型的SQL聚合。这也会引发我们对所用的存储方式以及最终的查询语言的关注。

4.1 Hadoop 分布式文件系统

Hadoop 所用的标准存储机制是 Hadoop 分布式文件系统（Hadoop distributed file system，HDFS）。它是以商用硬件为基础的，以实现容错。商用硬件的特性是，当有故障发生时，损坏的单元可以被替换。这是使用 RAID 存储的原因。但我们需要极其强大的可扩展性，使存储容量最高可达 PB 级。这也是比通常的 RAID 存储系统能够处理的更多的数据。

接下来假设我们要处理的是流数据，而不是随机数据访问。这些数据只是塞进磁盘，而由 RAID 系统控制数据的冗余。这是一个单次写入模型，假设数据在写入后永远不会改变。这种模型简化了数据的复制，并提升了数据吞吐量。但这意味着在数据进入系统前，前端不得不做好完备的验证和完整性检查。

习惯于 RDBMS（关系式数据库管理系统）的人讨厌这种缺乏数据完整性的情况。我们希望如数据库中一样，通过强制外键（FOREIGN KEY）约束来检查（CHECK()）约束和引用完整性。但是 HDFS 是一个文件系统，而不是一个数据库。大数据模型中我们可能会在最终获得数据的完整性。在此期间，我们假设我们可以在一定程度上接受不正确的和缺失数据的情况。

HDFS 是可跨操作系统移植的，但你会发现，Linux 是最流行的平台。这应该是毫不奇怪的，因为它在 Web 领域是被公认已久的。

庞大的数据量使程序越靠近数据速度就越快，HDFS 在这一点上具备相应的特性。HDFS 提供与常规文件系统类似的接口。它不像数据库那样，HDFS 只能存储和检索数据，但并不对数据建立索引。即便是对数据进行简单的随机访问也是不可能的。

4.2 查询语言

尽管使用原生的 API 可以连接到 HDFS，但开发人员更喜欢较高层次的接口。这样，他们就可以更快地编写代码，他们设计好这个过程，代码也可以移植到其他平台或编译器。

4.2.1 Pig Latin

Pig Latin，或简单称为 Pig，是由雅虎公司开发的，现在是 Hadoop 项目的一部分。它是针对使用工作流或有向图编程模型的开发者的，该模型可以并行运行，但每条“路径”又必须按顺序执行。

典型的 Pig 程序在开始执行时会有一个 LOAD 命令，并在最后有一个 STORE 命令。与其他过程式编程有很大区别的另一个特点是，赋值是永久性的，不能修改一个已经定义好的名

称。但是，可以重复使用它而不会丢失之前的对象。把每条语句想象为装配线上的资源站。你可以从源头取一条记录，把它传递到下一个资源站，然后再取下一条记录。下一个资源站将根据它所具有的数据来执行它的任务。

例如，记录中的一个字段是通过位置被引用的，使用美元符号和一个数字，从$0 开始。下面是来自维基百科文章中的关于 Pig 程序的一个例子。它通过从文本中逐行读取并过滤掉空白来提取单词。通过按单词分组，对各组进行计数，并将最终计数存入一个文件：

```
input_lines = LOAD '/tmp/my-copy-of-all-pages-on-internet' AS
(line:chararray);

-- Extract words from each line and put them into a pig bag
-- datatype, then flatten the bag to get Alpha word on each row
words = FOREACH input_lines GENERATE FLATTEN(TOKENIZE(line)) AS word;

-- FILTER out any words that are just white spaces
filtered_words = FILTER words BY word MATCHES '\w+';

-- create a GROUP for each word
word_Groups = GROUP filtered_words BY word;

-- count the entries in each GROUP
word_count = FOREACH word_Groups GENERATE COUNT(filtered_words) AS
count, GROUP AS word;

-- order the records BY count
ordered_word_count = ORDER word_count BY count DESC;
STORE ordered_word_count INTO '/tmp/number-of-words-on-internet';
```

开发人员可以用 Java 编写用户自定义函数（user-defined function，UDF）来获得更多的表现力。当然 SQL 也有 CREATE 函数可以使用外部语言，SQL 具备很强的表现力且足够强大，优秀的 SQL 程序员很少使用这种方式。

LOAD 命令是 SELECT 命令和 SQL 的数据定义语言（Data Declaration Language，DDL）之间的结合。它不仅仅用来获取记录，还有其他子句把该数据格式化为下一语句可以处理的内容。USING 子句会调用库中的过程，并用它从数据源中读取数据。AS (<字段对列表>)子句将记录分割成字段，并为它们分配一个相应的数据类型。字段对列表中的元素以逗号分隔，是(<字段名称>:<数据类型>)这样的对，也有可用于各种结构化数据的选项。DUMP 命令可以显示一个正在调试的对象的当前内容，它不会将其发送到持久存储中。除非你需要查看你的代码，否则不要担心它。

FILTER 命令是 Pig 版的 SQL 搜索条件。它使用 SQL 和 C 语言的符号、逻辑运算符，以

及等同于 SQL 的语义。例如：

```
Users_20 = FILTER Users BY age > 19 AND age < 30
```

在 BY 子句中谓词判断逻辑中可以包含 C 相等运算符==和！=而不是 SQL 的<>。其余的 θ 运算符[①]包括>、>=、<和<=。这些比较运算符可用于任何标量类型的数据，==和!=还可以应用到映射（map）和元组（tuple）这样的数据类型。要与 β 元组一起使用这些操作符，两个元组必须具有相同的模式（schema）或两者都没有模式。在袋类型中，上面的任何操作符都不适用。Pig 适用通常的运算符优先级和基本的数学函数，但它并不是一种计算语言，所以该语言并没有将巨大的数学函数库作为其一部分。

继 C 系列风格后，Pig 中的字符串也称为字符数组（chararrays），它具有 Java 的正则表达式语法和语义。由于字符数组是用 Java 编写的，因此这是毫无疑问的，但它却背离了 SQL 或其他语言程序员的习惯。SQL 是基于 POSIX 正则表达式的，其中有大量的简写。Perl 的正则表达式可以作用于字符串的一部分，而 Java 却不会。例如，如果你正在寻找包含字符串“Celko”的所有字段，就必须使用'.*Celko.*'，而不能使用'Celko'，前者才是完全匹配。只有 SQL 语言使用'%'和'_'作为通配符，其他语言都使用'.'表示单个字符匹配，'*'表示不定长度的字符匹配。

常用的逻辑操作 AND、OR 和 NOT 操作符都具有标准的优先级。Pig 有 SQL 的三元逻辑运算和 NULL，因此 UNKNOWN 在过滤器中将被视为 FALSE。Pig 在必要时可以按照短路逻辑[②]进行求值。这意味着 Pig 程序按照从左至右的顺序执行，而且在 FILTER 语句中一个谓词的值不受后续关系的影响时，求值就会停止。

由于 Pig 语言中允许使用用户自定义函数，并且其并不是函数式语言，这意味着一些代码可能不会执行，或者可能会产生副作用。举一个愚蠢的例子，假设有一个返回 TRUE 的用户自定义函数（UDF），但是在它返回之前，它在应用程序之外可能已经执行了一些其他的逻辑（如重新格式化所有硬盘驱动器）：

```
FILTER Foobar BY (1 == 2) AND Format_All_Drives_Function (x);
```

第一项(1 == 2)返回值是 FALSE，因此 Format_All_Drives_Function (x)将永远不会被调用，但是，如果我们把它写成下面这样，Format_All_Drives_Function (x)就会被调用了：

```
FILTER Foobar BY Format_All_Drives_Function (x) AND (1 == 2);
```

① 此处理解为比较运算符。——译者注

② 又称最小化求值，是一种逻辑运算符的求值策略。只有当第一个运算数的值无法确定逻辑运算的结果时，才对第二个运算数进行求值。参见：http://zh.wikipedia.org/wiki/短路求值。——译者注

如果没有人终止这个疯狂的操作，整个系统就会消失了。

副作用还可能会阻止其他优化行为，需要我们确定代码中不存在副作用。SQL/PSM通过要求过程或函数声明为不确定性，来解决这个问题。确定的函数在相同输入的情况下会返回相同的输出。如数学函数，如SIN()或COS()。现在重新想想，在一个文件中得到任何可能内容的下一条记录的FETCH语句恰好是不确定的。

有一个与Java程序中各种事物相关的网站。它太可爱了，以至于不能不推荐，这个网站是Piggybank[①]。这里面的软件包是根据函数的类型进行分类的。当前顶层包对应于功能类型，它们包括：

- org.apache.pig.piggybank.comparison——针对通过ORDER操作符进行的自定义比较；
- org.apache.pig.piggybank.evaluation——针对聚合和列转换这样的求值函数；
- org.apache.pig.piggybank.filtering——针对FILTER操作符中使用的函数；
- org.apache.pig.piggybank.grouping——针对分组函数；
- org.apache.pig.piggybank.storage——针对LOAD/STORE函数。

FOREACH语句会在数据管道的每个记录上应用相应的操作。经常写管道命令的Unix程序员会比较熟悉这种模式。FOREACH语句中输入一条名为Alpha的记录，然后输出一个Alpha记录来发送给流水线（pipeline）中的下一条语句。下一条语句将创建一条名为Beta的新记录。对于熟悉RDBMS的人来说，这是（在某种程度上）Pig实现关系投影运算的方式。例如，下面的代码加载一条完整的记录，但随后删除了除每条记录中的user_name和ID字段外的所有字段：

```
Alpha = LOAD 'input' as (user_name:chararray, user_id:long,
address:chararray, phone_nbr:chararray);
Beta = FOREACH Alpha GENERATE user_name, user_id;
```

但是，就“关系”的理念来理解，这并不是完美的投影。RDBMS是面向集合的，所以会一次完成全部数据的投影。Pig是一种工作流模型——我们会得到生成的元组流作为输出。这区别尽管非常微妙的，但很重要。

FOREACH有很多工具，其中最简单的是常量和字段引用。字段引用可以通过名称（SQL模型）或通过位置（面向记录的模型）引用。位置引用通过$和从0开始的数字执行，但我强烈反对位置引用，因为位置不记录执行过程。如果数据源发生变化，位置可能会发生改变。引用一个不存在的位置字段将返回NULL。引用在元组中不存在字段名将产生错误。

① 参见 https://cwiki.apache.org/confluence/display/PIG/PiggyBank。——译者注

使用描述性的名称和位置编号是为了安全，并且是最好的现代编程实践。如今，我们有文本编辑器可以使用，可以剪切和粘贴列表，不再使用 80 列打孔卡。例如：

```
Prices = LOAD 'NYSE_Daily_Ticker' as (exchange, symbol, date, open,
high, low, close, volume, adj_close);
Gain = FOREACH Prices GENERATE close - open; -- simple math
--Gain2 = FOREACH Prices GENERATE $6 - $3;
```

Gain 和 Gain2 将包含相同的值。除了使用的名称和位置进行引用外，还可以像 SQL 约定的那样使用*来引用一条记录中的所有字段。这将产生一个包含所有字段的元组。从 0.9 版本开始，你也可以使用 beta 周期以及[<start field>]..[<end field>]语法引用位于一个区间范围内的字段。这个语法按照声明的顺序扩展字段。如果没有明确的起始列，则使用第一个 α 列；如果没有明确的结束列，则使用最后的 α 列；否则，这个区间范围就会包括基于字段列表的开始字段和结束字段之间的所有字段。

非常有用的问号操作符，使用<谓词>? <真值>：<假值>的语法，对于比较老的 C 程序员是非常熟悉的。这是 Pig 版本 CASE 表达式的始祖！谓词被测试，如果谓词是 TRUE，表达式返回问号后面的“真值”。如果谓词是 FALSE，它就返回“假值”。这是其父语言 C 中问号操作符处理布尔逻辑的方式。但 Pig 有 NULL！准 SQL 的使用方式！这也许更容易从例子中看到：

```
2 == 2 ? 1 : 4 --returns 1
2 == 3 ? 1 : 4 --returns 4
NULL == 2 ? 1 : 4 -- returns NULL
2 == 2 ? 1 : 'Celko' -- type error, string vs integer
```

Pig 有一个“袋”（bag）的概念，SQL 是基于其上的。它是无序的元组的集合，并且允许在集合中存在重复。但是，SQL——优秀的 SQL——会通过 PRIMARY KEY 来保证袋中元组是一个真集。

GROUP 语句并不是 SQL 语句中使用的 GROUP BY！SQL 中的 GROUP BY 是返回表的汇总语句。Pig 的 GROUP 语句将具有相同键的记录收集在一起，其结果并不是汇总的数据，这条语句是构建“袋”集合的中间步骤。在 Pig 中，如果需要，可以应用聚合函数。例如：

```
Daily_Ticker = LOAD 'Daily_Stock_Prices' AS (stock_sym, stock_price);
Daily_Stock_Groups = GROUP Daily_Ticker BY stock_sym;
TickerCnt = FOREACH Daily_Stock_Groups GENERATE GROUP, COUNT(Daily
Ticker);
```

这个例子是按股票的股票代码对记录进行分组，然后再统计它们。从 GROUP BY 语句中输出的数据有两个字段：键和收集的记录的袋。键字段通过 GROUP 命名，袋则以被分组数据的别名命名。所以，在前面的例子中，它将被命名为 Daily_Ticker 并继承 Daily_Ticker

的模式。如果 Daily_Ticker 关系没有模式，则名为 Daily_Ticker 的袋也没有模式。对于 GROUP 中的每条记录，整个记录，包括键，都在袋中。

还可以对多个键使用 GROUP。键必须在括号中，就像 SQL 中的行构造函数。但不同于 SQL，我们可以在结果中使用元组作为字段；因此，我们仍然需要让记录有两个字段，但这里的字段会比 SQL 中的标量列更加复杂。

在这一点上，很容易用一个例子来说明。让我们建立两个数据集，分别为 Alpha 和 Beta：

```
Alpha = LOAD 'Alpha' USING PigStorage();
Beta = LOAD 'Beta' USING PigStorage();
```

PigStorage 是一个标准库函数，可以让我们从一个标准源读取记录。假设数据是下面这样的：

```
Alpha:
a     A    1
b     B    2
c     C    3
a     AA   11
a     AAA  111
b     BB   22

Beta:
x     X    a
y     Y    b
x     XX   b
z     Z    c
```

现在，我们可以使用一些类似于关系数据库的奇特的命令。我们已经讨论过 GENERATE，但这里主要介绍它是如何被转储的。注意括号以及零初始位置的使用：

```
Alpha_0_2 = FOREACH Alpha GENERATE $0, $2;
(a, 1)
(b, 2)
(c, 3)
(a, 11)
(a, 111)
(b, 22)
```

这里的 GROUP 语句会首先显示分组键，然后将元组行显示在花括号中。数学专业的学生可能会为此很高兴，因为花括号是一个枚举集的标准符号。还请注意，字段还是按照原来的顺序排列的：

```
Alpha_Grp_0 = GROUP Alpha BY $0;

(a, {(a, A, 1), (a, AA, 11), (a, AAA, 111)})
```

```
(b, {(b, B, 2), (b, BB, 22)})
(c, {(c, C, 3)})
```

当分组键是多个字段时，该行的构造数据在圆括号中，但字段的列表仍然在花括号内：

```
Alpha_Grp_0_1 = GROUP Alpha BY ($0, $1);

((a, A), {(a, A, 1)})
((a, AA), {(a, AA, 11)})
((a, AAA), {(a, AAA, 111)})
((b, B), {(b, B, 2)})
((b, BB), {(b, BB, 22)})
((c, C), {(c, C, 3)})
```

Pig 有 3 个基本聚合函数，它们看起来像 SQL：SUM()、COUNT()和 AVG()。这些取整和展示规则并不奇怪，但 Pig 并没有所有这些年来已经被添加进 ANSI/ISO 的标准 SQL 中的常用 SQL 操作符，例如：

```
Alpha_Grp_0_Sum = FOREACH Alpha_Grp_0 GENERATE GROUP, SUM(Alpha.$2);

(a, 123.0)
(b, 24.0)
(c, 3.0)

Alpha_Grp_0_Cnt=FOREACH Alpha_Grp_0 GENERATE GROUP, COUNT(Alpha);

(a, 3)
(b, 2)
(c, 1)

Alpha_Grp_0_Avg=FOREACH Alpha_Grp_0 GENERATE GROUP, AVG(Alpha);

(a, 41.0)
(b, 12.0)
(c, 3.0)
```

现在我们开始了解一些更有趣的内容！LISP 程序员会很熟悉 FLATTEN，LISP 中提供了具有相同名称的函数。它以花括号括起来的元组作为输入，并将它们放入一个列表。这就是为什么在元组要拥有键是很重要的，你并没有破坏数据本身。例如：

```
Alpha_Grp_0_Flat = FOREACH Alpha_Grp_0 GENERATE FLATTEN(Alpha);
(a, A, 1)
(a, AA, 11)
(a, AAA, 111)
(b, B, 2)
```

```
(b, BB, 22)
(c, C, 3)
```

COGROUP 是一种连接操作。可以在有 3 个或更多字段的数据中使用。COGROUP 参数中最前面的是什么样的值，随后的元组也应该是一样的。每个 BY 子句告诉你使用元组的哪一列，NULL 被视为跟任何数据都相等的，就像我们在 SQL 中的分组运算符中做的一样。例如：

```
Alpha_Beta_Cogrp = COGROUP Alpha BY $0, Beta BY $2;

(a, {(a, A, 1), (a, AA, 11), (a, AAA, 111)}, {(x, X, a)})
(b, {(b, B, 2), (b, BB, 22)}, {(y, Y, b), (x, XX, b)})
(c, {(c, C, 3)}, {(z, Z, c)})
```

再注意一下，这是一个嵌套结构。Pig 的风格是建立一个操作步骤的链，使引擎可以在工作流模型中发挥并行的优势。但是，这往往意味着需要取消这些嵌套结构。看下面这个例子，并研究它：

```
Alpha_Beta_Cogrp_Flat = FOREACH Alpha_Beta_Cogrp GENERATE
FLATTEN(Alpha.($0, $2)), FLATTEN(Beta.$1);

(a, 1, X)
(a, 11, X)
(a, 111, X)
(b, 2, Y)
(b, 22, Y)
(b, 2, XX)
(b, 22, XX)
(c, 3, Z)
```

JOIN 是经典的关系自然等值连接，但是在 SQL 将冗余连接列从结果表中去除时，Pig 将会保持两者同时存在。下面这个例子将连接字段放在行的结尾，因此可以在结果中看到它们。同时还要注意，Alpha 和 Beta 是如何保持其各自的一致性的，$位置表示法并没有应用在结果中：

```
Alpha_Beta_Join = JOIN Alpha BY $0, Beta BY $2;

(a, A, 1, x, X, a)
(a, AA, 11, x, X, a)
(a, AAA, 111, x, X, a)
(b, B, 2, y, Y, b)
(b, BB, 22, y, Y, b)
(b, B, 2, x, XX, b)
(b, BB, 22, x, XX, b)
(c, C, 3, z, Z, c)
```

如果你喜欢经典的集合论就会知道，CROSS 是经典的关系交叉连接或笛卡儿积。对于 SQL 程序员来讲这可能是很危险的。在 SQL 中，SELECT .. FROM ..语句被定义为一个在 FROM 子句中的交叉连接，并投影到 SELECT 子句中。在现实世界中没有哪款 SQL 引擎实际上按照这种方式来实现，但由于 Pig 是一款按部就班的语言，完全可以做到这一点！从本质上讲，Pig 程序员必须是自己来完成所需的代码优化。例如：

```
Alpha_Beta_Cross=CROSS Alpha, Beta;

(a, AA, 11, z, Z, c)
(a, AA, 11, x, XX, b)
(a, AA, 11, y, Y, b)
(a, AA, 11, x, X, a)
(c, C, 3, z, Z, c)
(c, C, 3, x, XX, b)
(c, C, 3, y, Y, b)
(c, C, 3, x, X, a)
(b, BB, 22, z, Z, c)
(b, BB, 22, x, XX, b)
(b, BB, 22, y, Y, b)
(b, BB, 22, x, X, a)
(a, AAA, 111, x, XX, b)
(b, B, 2, x, XX, b)
(a, AAA, 111, z, Z, c)
(b, B, 2, z, Z, c)
(a, AAA, 111, y, Y, b)
(b, B, 2, y, Y, b)
(b, B, 2, x, X, a)
(a, AAA, 111, x, X, a)
(a, A, 1, z, Z, c)
(a, A, 1, x, XX, b)
(a, A, 1, y, Y, b)
(a, A, 1, x, X, a)
```

SPLIT 是 Codd 博士的原始关系运算符。它从来没有流行起来，因为它返回两个表，而且这个功能可以通过其他关系运算符（Maier，1983，p37-38）来完成。但 Pig 提供了它的一个实现，可以让你将数据拆分至一条语句中的几个不同的“桶”（bucket）中，具体如下：

```
SPLIT Alpha INTO Alpha_Under IF $2<10, Alpha_Over IF $2>= 10;
-- Alpha_Under:
(a, A, 1)
(b, B, 2)
(c, C, 3)
```

```
-- Alpha_Over:
(a, AA, 11)
(a, AAA, 111)
(b, BB, 22)
```

你是否注意到，可能有一些行没有放入“桶”中？

在这个模型中有一个折中。在SQL中，优化器能够统计并使用这些统计数据来创建一个执行计划。如果改变了统计结果，执行计划就会改变。在Pig或基于Web的环境中没有办法收集统计信息。一旦程序被写好，就必须使用并忍受它。

这也意味着，没有办法均匀分配工作负荷以实现其均匀地分布在系统中的reducer上。如果其中一个单元正在承担一项巨大的工作负荷，其他单元都要等，直到工作流中的所有单元都已经准备好传递数据到下一个步骤。事实上，靠一个reducer来管理这么多的数据或许是不可能的。

Hadoop有一个combiner阶段，这个阶段不删除任何倾斜数据，但会在相应的数据上放置一个绑定（打一个标记）。并且，大多数的任务中，mapper的数量最多可能是数万，即使reducer得到了倾斜记录，每个reducer上得到的记录的绝对数量也会足够小，以便reducer能够快速地处理它们。

一些像SUM这样的计算可以分解成任意数目的步骤，称为可分配的，它们与combiner一起工作。还记得高中的代数课程吗？这就是分配律[①]，我们喜欢它。

可以分解成一个初始步骤、任何数量的中间步骤以及一个最终步骤的计算称为代数。可分配的计算是代数特性中的特殊情况，其中的初始步骤、中间步骤和最终步骤都是相同的。COUNT是这种函数的一个例子，其中的初始步骤是一个计数，中间步骤和最终步骤是各个计数的总和（附加计数）。中位数不是代数，在找到中间值之前，必须将所有记录按照某个/某些字段进行排序。

Pig的真正工作是用来构建尽可能多的使用combiner的用户自定义函数，因为它的倾斜数据归并特性，以及早期的聚合大大减少了需要网络传输以及写入磁盘的数据量，从而加速性能显著。这是很不容易的，所以我就不去介绍它了。

4.2.2 Hive和其他工具

正如Pig是雅虎的项目，Hive是产生于Facebook的一个开源的Hadoop语言。它比Pig更接近于SQL，它像Pig一样，不被编译就可以用于即时查询。它是包括Cassandra和Hypertable等产品系列的代表。它们都使用HDFS作为存储系统，但在HDFS上使用基于表的抽象，所

① 在抽象代数中，分配律是二元运算的一个性质，它是基本代数中的分配律的推广。参见 http://zh.wikipedia.org/wiki/分配律。——译者注

以容易装入结构化数据。Hive QL 是执行 MapReduce 作业的类 SQL 查询语言。但是，它可以使用 Sqoop 从关系数据库中将数据导入 Hadoop。它是由 Cloudera 为自己的 Hadoop 平台产品开发的。Sqoop 是数据库无关的，因为它使用了 Java JDBC 数据库 API。表可以整体导入，也可以使用查询来限制数据导入。Sqoop 还提供了将来自 HDFS 的 MapReduce 的结果重新注回一个关系数据库的能力。这意味着，Hive 是用于分析的，而不是为了联机事务处理（OLTP）或批处理。

你可以如同在 SQL 中那样用一组简单的数据类型（INT、FLOAT、DATE、STRING 和 BOOLEAN）来声明带有列的表。真正强大的能力同样来自使用简单的数据结构。

- 结构体：结构体类型内的元素可以使用点符号（.）访问。例如，对于 `STRUCT {a INT; b INT}`类型的列 `c` 中，`a` 这个字段可以通过表达式 `c.a` 进行访问。
- 映射（键值元组）：内部的元素使用`['元素名']`进行访问。例如，在一个由来自 `'group'→gid` 的映射关系组成的映射 `M` 中，`gid` 的值可以使用 `M ['group']`进行访问。
- 数组（可索引的一维列表）：该数组中的元素必须是同一类型的。在数组中，元素可以使用[*n*]符号进行访问，其中 *n* 是一个索引（从 0 开始）。例如，对于一个具有元素 `['a', 'b', 'c']`的数组 `A`，`A[1]`返回`'b'`：

```
CREATE TABLE Foo
(something_string STRING,
 something_float FLOAT,1
 my_array ARRAY<MAP<['foobar'],
 STRUCT <p1:INT, p2:INT>>);

SELECT something_string, something_float,
    my_array[0], ['foobar'].p1
 FROM Foo;
```

默认的情况下表中的文本字段由 Ctrl+A 标记进行分割，记录之间由换行符进行分割。可以添加更多的子句来定义分隔符和文件布局。例如，一个简单的 CSV 文件可以通过添加以下内容到 `CREATE TABLE` 定义的结尾来定义：

```
ROW FORMAT DELIMITED
 FIELDS TERMINATED BY '\,'
 STORED AS TEXTFILE
```

声明的另一个重要部分是，一个表可以在存储上进行分区，并且你可以通过分区操作表。例如，可以通过一条子句从这些分区中获取随机样本：

```
TABLESAMPLE(BUCKET x OUT OF y)
```

SELECT 也可以使用中缀连接，但它只允许等值连接。可以使用 INNER JOIN、LEFT OUTER、RIGHT OUTER 和 FULL OUTER 等语法。

还有一个 LEFT SEMI JOIN，以检查一个表是否引用了另一个表；如果使用了 SQL 子集，这个功能是无法完成的。编译器做不到用于在 SQL 中的同样的优化，因此要把最大的表放到连接的最右侧，以获得最佳性能。按照惯例，还有 UNION ALL 操作，但没有其他集合操作符了。

Hive 也使用了通常的 SQL 聚合函数。其他内置函数混合使用 SQL 和 C 系列的语法，经常会有两个选项，也就是说，UPPER() 和 UCASE 是相同的功能，只是名字不同，等等。

总结思考

如果你能够理解 SQL 的执行计划，对数据的 MapReduce 模型的基本概念的理解就应该不会有任何问题。你的学习曲线将随着不得不使用 Java 和其他底层工具而上升，这些工具是 SQL 编译器用来完成复杂任务的一部分。

缺乏一个包含统计、事务层次并内置数据完整性的 SQL 风格的优化器，（如果能解决）将是一个重大飞跃。你会发现，你必须手动做这些事情，并且不得不做。

成熟的编程语言有风格和规范，就像人类的语言一样。遇上同样的问题，COBOL 程序员解决问题的方式同 SQL 程序员解决问题的方式是有很大差别的。MapReduce 还在发展它自己的风格和规范。事实上，它仍然试图找到一种标准语言，以使其能匹配 SQL 在 RDBMS 的位置。

参考文献

Maier, D. (1983). *Theory of relational databases*. Rockville. F. MD: Computer Science Press. ISBN: 978-0914894421.

Capriolo, E., & Wampler, D. (2012). *Programming hive*. Cambridge: O'Reilly Media. ISBN: 978-1449319335.

Gates, A. (2012). *Programming Pig*. Cambridge, MA: O'Reilly Media. ISBN: 978-1449302641.

流式数据库和复杂事件

简介

本章讨论流式数据库。这些工具都在关注随着时间流动的数据，并且这些数据在系统中流动时必须被“抓取”到。与静态数据不同，我们需要寻找这个临时流动中的事件的模式。这依然不是数据的计算模型。

关系数据库模型假定在一个查询过程中表是静态的，并且结果也是静态表。如果你想在大脑中有一个直观的印象，可以认为在查询或其他数据操纵语言（Data Manipulation Language，DML）语句使用该数据的时候，数据库就是一个水库。笼统地来讲，这是传统的关系数据库系统（RDBMS）。

但是，流式数据库是建立在不断流动的数据上的，可以认为是河流或是数据的管道。流式数据模型中最有名的例子是由软件在亚秒级别就要完成交易（指非常快速地完成）的股票和商品交易。究其原因，众所周知的是，如果这些应用程序陷入困境，它们就会上报纸。继续用水/数据的比喻进行类推，如果管道破裂或厕所暂停使用，每个人都会知道。

传统的关系数据库系统（RDBMS）就像一个静态的水库，关注数据容量。厂商们常常谈论他们的产品可以处理的数据量达到 TB 级（最近已经达到 PB 级或 EB 级）。在他们的口中，交易量和响应时间都是次要的。他们做广告以及基准测试的前提是，传统的 OLTP 系统使用现有的技术已经做得“足够好”。

流式数据库，像一个消防水带，关注数据量，但更重要的是流速、速度和速率。有一个浴缸，一个排水管以每分钟 10 升和每分钟 100 升的速度给其注水是有根本区别的。同样，使用工业压力切割器粗管和窄喷嘴在速率高达每秒 100 升的时候也是有差别的。

5.1　代并发模型

数据库必须支持多用户并发。共享数据是使用 DBMS 的一个重要原因，单个用户最好使用文件系统。但是，这意味着我们的系统中需要某种“流量控制”。

这个概念来自不得不管理硬件的操作系统。两个作业不应在同一时间写同一打印机，因为它会产生垃圾打印输出。多个会话同时从 RDBMS 中读取数据并不是大问题，但是两个会话要同时在同一个表中更新、删除或插入数据就不是那么容易了。

5.1.1　乐观并发

乐观并发控制假定冲突是例外情况，并且在其发生后我们必须对其进行处理。乐观并发的这一模型的模拟场景是缩微胶片。如今大多从事数据库工作的人都没有看过缩微胶片，所以，如果你没看过，你可能要用 Google 搜索一下。这种方法比数据库早几十年。它是在开始在缩微胶片上存储数据时，由公司的中央记录部门手动实现的。用户不会拿到缩微胶片的原始版本，而是由影片管理员提供一份做了时间戳标记的副本给他。用户把副本放到自己的办公桌上，打好标记，并将其返还给中央记录部门。中央记录员为更新的文件打好时间戳，将其拍照，并把它添加到缩微胶片的末端。

但是，如果第二个用户，即用户 B，也去了中央记录部门，并得到了同一份文档的带有时间戳的副本，情况会怎样呢？中央记录部门的员工就不得不查看两份时间戳，并作出决定。如果用户 A 试图把他的更改更新到数据库中，而用户 B 仍然在她的副本上进行修改，那么中央记录部门的员工就不得不要么保持持有的第一个副本，等待第二个副本（等待 B 完成），要么将第二个副本返回给用户 A。如果两个副本都已经完成（A 和 B 都已经返还），中央记录部门的员工就会把这两份副本分别放到 A 和 B 两个位置，在光线下仔细查看，以检查是否有任何冲突。如果这两份被修改的副本不存在冲突，中央记录部门的员工可以直接入库；如果发现有冲突，中央记录部门的员工就必须有解决问题的规则或是两个都拒绝。这种行为代表了一种在冲突发生后才会进行的行级锁。

副本上有一个时间戳，称为 t_0 或 `start_timestamp`。变化是通过将数据的新版本加到带有时间戳 t_1 的文件的结尾来提交的。时间戳在系统中是唯一的。由于现代机器能够支持纳秒，不需要连续的顺序编号，实际的时间戳就可以完成工作。如果你想使用并了解这种模型，可以获取 Borland 的 Interbase 或是其开源产品 Firebird 的副本。

5.1.2　乐观并发下的隔离级别

数据私有副本上运行的事务是永远不会被阻塞的。但是，这意味着，在任何时刻，由主

动、已提交的事务创建的每个数据项都可能具有多个版本。

当事务 T_1 准备提交时，它会得到一个 commit_timestamp，这个 commit_timestamp 晚于任何现有的 start_timestamp 或 commit_timestamp。如果本次事务提交成功，则必须满足：没有任何其他事务（也就是说 T_2），其 commit_timestamp 在 T_1 的[start_timestamp, commit_timestamp]时间区间内写 T_1 也在写的数据。否则，T_1 将回滚（ROLLBACK）。这种“第一个提交者胜利”的策略可以防止更新丢失（现象 P4）。当 T_1 提交后，T_1 带来的数据变化会对那些 start_timestamps 比 T_1 的 commit_timestamp 大的所有事务可见。这就是所谓的*快照隔离*（snapshot isolation），但它也有其自身的问题。

快照隔离是非序列化的，因为事务的读操作出现在某一瞬间，而写出现在另一瞬间。假设你有几个作用在同一数据上的事务，并且这些事务在表中的约束为（$x + y > 0$）。每一个为 x 和 y 写新值的事务都必须遵守该约束。而 T_1 和 T_2 在它们各自的副本上进行隔离时，其表现均正常，但如果将两个事务放在一起时，其约束就会失败。可能出现的问题有以下几个。

- A5（*数据项违反约束*）：假设约束 C 是数据库中两个数据项 x 和 y 之间的数据库约束。有两种因违反约束而产生的异常。
- A5A（*读倾斜*）：假设事务 T_1 读了 x，然后 T_2 更新了 x 和 y（x 和 y 均为新值）并且提交了。现在，如果 T_1 读 y，它可能会看到不一致的状态，因此会产生一个不一致的状态作为输出。
- A5B（*写倾斜*）：假设 T_1 读 x 和 y，并与约束 C 相符，然后 T_2 读 x 和 y，写 x 并且提交。然后 T_1 写 y，如果 x 和 y 之间有约束，那么约束可能就会被违反。
- P2（*模糊读*）：这是 A5A 在 $x = y$ 的情形下的退化形式。更典型的情况是，一个事务读两个不同但相关的数据项（如参照完整性）。
- A5B（*写倾斜*）：这是一个可能在银行系统中产生的约束。在银行账户中，只要通常持有的余额的总和仍是非负数，账户的余额允许为负数[①]。

显然，在 P2 被排除的情况下，A5A 和 A5B 都不可能出现，这是因为 A5A 和 A5B 都需要 T_2 写之前 T_1 读过但未提交的数据项。因此，现象 A5A 和 A5B 仅具有区分 REPEATABLE READ 以下隔离级别的实力。

在其严谨的解释中，ANSI SQL 定义的 REPEATABLE READ 捕捉了行约束的退化形式，但却忽略了一般概念。具体来说，用事务级别的 REPEATABLE READ 来锁定一个表，为行提供了阻止违反行约束的保护，但禁止出现异常 A1 和 A2 的 ANSI SQL 定义却没有提供这种保

① 参见 http://zh.wikipedia.org/wiki/国际收支。——译者注

护。快照隔离甚至比 READ COMMITTED 更强（见表 5-1）。

表 5-1　根据 3 个初始现象定义的 ANSI SQL 隔离级别

隔离级别	P0（或 A0）脏写	P1（或 A1）脏读	P2（或 A2）模糊读	P3（或 A3）现象
READ UNCOMMITTED	不可能	可能	可能	可能
READ COMMITTED	不可能	不可能	可能	可能
REPEATABLE READ	不可能	不可能	不可能	可能
SERIALIZABLE	不可能	不可能	不可能	不可能

这里最重要的属性是你可以在时间戳 t_n 读取数据，同时可以在时间戳 t_{n+k} 并行更改数据。在“流”的不同部分读取数据，可以重新构建数据库在过去的任何时间点的一致视图（见表 5-2）。

表 5-2　一致性程度及根据锁定义的锁隔离级别

一致性级别=加锁隔离级别	数据项及谓词上的读锁（相同，除非特别标注）	数据项及谓词上的写锁（总是相同）
度 0	没有要求	写良好
度 1=加锁 读未提交	没有要求	写良好，长时间写锁
度 2=加锁 读已提交	读良好，短时间读锁（双方） 读良好	写良好，长时间写锁
游标固定	读锁保持在当前游标 短时间读谓词锁	写良好，长时间写锁
锁 可重复读	读良好 长时间数据项读锁 短时间读谓词锁	写良好，长时间写锁
度 3=序列化锁定	读良好，长时间读锁（双方）	写良好，长时间写锁

5.2　复杂事件处理

传统的 RDBMS 的另一假设是，在查询时，在数据模型上的约束是始终有效的。所有的水是在同一个地方，并随时可以饮用。但流式数据模型与复杂事件处理（complex event processing，CEP）有关。这意味着，并非所有的数据都已经到达数据库了！你还不能完成你的查询，因为当前还只有你预期数据的一部分。这些数据可以是来自同一数据源，也可以来自多个数据源。

事件的概念通过一篇想象你在咖啡店里的文章来解释是很容易的。有一些咖啡店是同步的：一位客人走到柜台并点好咖啡和糕点，柜台后面的服务员把糕点放进微波炉，并开始准备咖啡，把糕点从微波炉里面取出来，收钱，并把托盘交给客人，然后转向服务于下一位客人。每位客人是一个单独的线程。

一些咖啡店是异步的：在柜台后面的服务员接收订单和收款，然后接着服务下一个客户，快餐厨师负责加热糕点，咖啡师只做咖啡。当咖啡和糕点都准备好了，他们会招呼客人到服务台来取或是根据订单将餐品送到客人的桌上。客户可以坐在那里，拿出一台笔记本电脑，边写书边等待。

该模型具有生产者和消费者，或者输入流和输出流。加热后的糕点对于食用它的消费者来讲就是输出。如果你愿意的话，该事件可以在数据流图中进行建模。

在这两类店中，这是预期行为。现在想象一下，一个强盗进入咖啡厅，盗走了客人的钱财。这不是常规的预期行为。受害人不得不向警方报案，并可能会要求信用卡公司取消被盗的信用卡。这些事件将触发进一步的事件，如法院的出现、激活新的信用卡等。

5.2.1 与事件处理相关的术语

当我们谈论时总是会讨论一些术语，这是很不错的，所以让我们讨论一些。有一种情况是，事件是需要响应的。在咖啡店的例子中，纸杯不足就是一个事件，但这个事件可能并不需要立即响应。（“现在纸杯是什么情况？余量已经很少了！”）但是，纸杯已经用完这样的事件就需要立即的响应，因为没有纸杯就不能卖咖啡了。这种模式就是“检测→判断→响应”，无论是人还是机器。

（1）观察：事件处理用于对系统或进程的监控，通过寻找异常行为，并在这些行为发生时生成警报。在这种情况下，如果有后续的操作，警报的接收者需要及时进行反应；事件处理程序的职责只是产生警报。

（2）信息传播：当一个事件被观察到（发现），系统必须以合适的粒度在正确的时间内将正确的信息传递给对应的信息接收者。这是一个个性化的信息传递。我最喜欢的是收到银行关于我的支票账户的电子邮件。有一家银行会让我设置每日的限额，如果超过限额，就会提醒并且只会提醒我自己。另一家银行会每一天给我发送一个简短的说明，内容是关于前一天我银行账号交易的信息。他们不告诉我妻子我的支出或存入，只有我。

（3）动态操作行为：在这个模型中，系统的操作是由输入事件自动驱动的。在线交易系统使用了这种模型。遗憾的是，系统并不一定要有良好的判断力，因此需要有一种方法来防止无休止的反馈。正是出于这种原因，2011 年引发了亚马逊网上书店出现了价格为 23 698 655.93 美元的一本关于“苍蝇”的书。下面是事件的详细情节：Peter Lawrence 的 *The Making of a Fly*

是一本关于苍蝇的生物学著作，出版于 1992 年，已经绝版。但亚马逊上列出了 17 本进行出售：15 本是二手书，价格是 35.54 美元，另外 2 本分别来自两家合法的书商。书的价格在几天内不断上升，并慢慢汇聚成一个模式。由 Bordeebook 提供的书的价格是由 Profnath 提供的书价格的 1.270589 倍。Profnath 每天都设置它们的价格是 Bordeebook 价格的 0.9983 倍。这样的价格密切地保持几个小时，直到 Bordeebook“注意到”Profnath 的价格变化，然后再提升他们的价格，调整到 Profnath 更高的价格的 1.270589 倍。

亚马逊零售商越来越多地使用算法进行定价（亚马逊的东西本身规模就很大），有多家企业通过对零售商价格进行全面的检查，然后提供定价算法/服务。上文所述的 Profnath 使用的算法希望使他们的价格是最低的，但是仅一点点差距就会将他们的书目放到排序清淡的顶部。很多业务（购买行为）来自前三的位置！

在另一方面，Bordeebook 有很多积极的反馈，所以他们的算法会认为，买家会为他们的信心支付溢价[①]。后来，我拥有自己的书店后发现，确实有“书探”这种群体，他们通过为客户寻找书目和书店并为客户买书而生存。Bordeebook 的算法似乎是“书探”的机器人版本，其操控书的价格始终为 Profnath 价格的 1.270589 倍。

（4）*主动诊断*：该事件处理应用程序通过观察症状来诊断问题。对我们大多数人来讲，呼叫台是最常见的例子。关于世界上第一个呼叫台有一个经典的互联网笑话（http://www.why-not.com/jokes.old/archive/1998/january/27.html），其对话开头如下：“这里是火灾救援。我叫 Groog。”“我叫 Lorto。请帮帮我，我这里打不着火。”Grogg 恨不得用大棒子打 Lorto 一顿，这回复实在是太愚蠢了。

制造系统基于观察到的症状来查找产品的故障。尽管眼前的目标是要纠正的制造过程中的缺陷，但真正的目标是要找到引发这些症状的根本原因。如果将制造视作“流”，你将了解到一个统计抽样的学科，称为序列分析，是 Abraham Wald（1973）为了做弹药测试而发明的。（基本的想法是，测试的子弹、灯泡和许多其他产品是具有破坏性的，因此我们想从流中获取能够给予我们所想要的置信水平的最小样本量。但样本大小并不一定必须是常量！如果样本中有大量失败的结果，那么我们就需要增加样本量；如果失败率很低，就可以减少测试样本量。如果你想做更多的研究，可以使用称为 Haldanes 的逆抽样的技术来进行调整。

这个想法其实可以追溯到统计初期，在文献中，有一个时髦的名字叫“赌徒的破产[②]”（gambler's ruin）。一个赌徒如果每次赢了后都把自己的赌注增加到其全部赌本的一个固定份

① bordeebook 商户为什么要比最高价高 1.27 倍呢？合理的解释是 Bordeebook 并没有这本书，这个商户只是想用更多的选品来吸引买家，这样可以让人觉得他和竞争对手有一样多的选品。所以，他要把价订得高一点，这样就算是被人下单，他可以从别人手里把书买过来，然后再卖给卖家。27%的空间，够他赚了。参见 http://coolshell.cn/articles/4605.html。——译者注

② 参见 http://zh.wikipedia.org/wiki/赌徒破产理论。——译者注

额，但是在输了以后又不减少其赌本，那么即使他在每次下赌注都有一个正的期望值，他最终也将破产。有一些问题，只能通过顺序分析的方式解决。

（5）预测处理：你可能希望在一些事件发生前找出它们，这样就可以提前消除，避免其后果；事实上，这些事件可能根本就不存在！想象一下，一个金融机构的欺诈检测系统的检测模式。当它怀疑有欺诈行为发生时，它就会采取行动，但可能产生误报，因此在确定欺诈是否确实发生之前，需要做进一步的调查。金融行业中的经典例子是不放假的员工（这名员工可能是贪污犯）、突然申请了很多信用卡的客户（这个客户可能要破产）、客户的购买模式发生了变化（可能是被盗用了身份）。

这些不同类别的事件并不相互排斥，因此应用程序可以分为几类别。

5.2.2 事件处理与状态更改约束

事件模型与系统状态转换模型是不太一样的。状态转是完整性检查，为了保证数据仅根据规则序列变化，或者仅根据固定的或可变的生命周期、授权、商业计划、标书等的变化。可以在 SQL 中使用数据定义语言（Data Declaration Language，DDL）约束，通过一个辅助表来保证状态转换满足约束。这里不详谈，因为这更多是 SQL 编程的话题。

这样的约束可以通过状态转换图进行建模，以强制一个实体仅可按照特定方式进行规则的更新。有一个初始状态、显示下一个合法状态的流程线和一个或多个终止状态。最原始的例子是可能的婚姻状态的简单状态变化图，如图 5-1 所示。

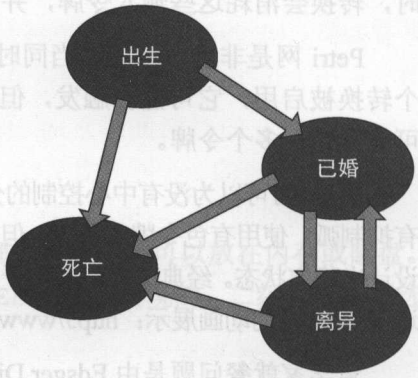


图 5-1 婚姻状态转换

下面是一个 DDL 框架，包含所需的 FOREIGN KEY，以引用有效的状态变化，以及与我相关的当前状态开始的日期：

```
CREATE TABLE MyLife
(previous_state VARCHAR(10) NOT NULL,
 current_state VARCHAR(10) NOT NULL,
 CONSTRAINT Valid_State_Change
 FOREIGN KEY (previous_state, current_state)
 REFERENCES StateChanges (previous_state, current_state),
 start_date DATE NOT NULL PRIMARY KEY,
 --etc.);
```

这些都是被锁定在僵化模式的状态。在这个例子中初始状态是“出生”，终止状态是“死亡”，这是一个必然的终止状态。这其中有隐含的时间顺序，但没有时间戳来指明它们发生的

时间。橡子变成木材以及最后变成抽屉前，要先成长为一棵橡树。橡子不会跳变成为一个抽屉。这是一种约束，而不是一个事件，其本身并没有任何行为。

5.2.3　事件处理与 Petri 网

Petri 网是一个用来解决 CEP 问题的系统的数学模型。有大量关于 Petri 网的研究，它们被用来设计具有复杂计时问题的计算机系统。一个 Petri 网由库所[①]（表示为圆圈）、变迁（表示为线或条状[②]）和弧（有向箭头）组成。

弧从一个库所连接到一个变迁，或者反之，但绝不在库所之间或是变迁之间进行连接。弧连接到一个变迁的库所称为转换的输入库所，有向弧从一个变迁连接到的库所称为变迁的输出库所。

从图形上看，Petri 网内包含零个或多个令牌（显示为黑点）。转换触发时令牌将围绕 Petri 网移动。当输入狐的起点处具有足够多的令牌时，Petri 网的转换就可能会触发。在转换发起时，转换会消耗这些输入令牌，并用原子步骤在输出弧末端的库所放置新令牌。

Petri 网是非确定性的：当同时启用多个转换时，它们中的任何一个都可能触发。如果一个转换被启用，它可能会触发，但并不是必须的。在网络的任何地方（甚至在同一位置）都可能会存在多个令牌。

Petri 网可以为没有中心控制的分布式系统中的并发行为进行建模。这种技术的实践版本中会有抑制弧，使用有色令牌，等等。但重要的一点是，你可以证明 Petri 网可以从任何初始标识开始，设计出稳定状态。经典的教科书例子是双向红绿灯和哲学家就餐问题，你可以在下面的网址中看到这些 Petri 网的动画展示：http://www.informatik.uni-hamburg.de/TGI/PetriNets/introductions/aalst/。

哲学家就餐问题是由 Edsger Dijkstra 设计的，作为 1965 年的学生考试题目，但 Tony Hoare 给了我们这个题目的当前版本：5 位哲学家坐在一张桌子旁，在他们每个人面前有一碗米饭。每对相邻的哲学家之间放有一根筷子。每个哲学家必须交替思考和吃饭。但是，哲学家只能在其有一双筷子时吃米饭。每根筷子只能由一位哲学家持有，所以一位哲学家只能在一根筷子不被另一位哲学家使用时才能使用。在一位哲学家吃完后，需要放下两根筷子将它们提供给他人。哲学家可以在其左边或右边拿到筷子，但在拿到两根筷子前不能开始吃饭。这里，假设饭碗总是装满米饭的。

没有任何一位哲学家能够有什么办法知道其他人会如何做。问题是如何保持哲学家们的行为在饮食和思考之间永远交替，而没有任何人饿死。

---

① 参见 https://zh.wikipedia.org/wiki/Petri 网。——译者注

② 原文错误，应表示为方形节点。——译者注

5.3 商业产品

你可以从IBM（SPADE）、甲骨文（Oracle CEP）、微软（StreamInsight），以及一些较小的厂商，如StreamBase（SQL的面向流的扩展）和Kx（Q语言，基于APL）获取流式数据库的商业产品，以及一些开源项目（Esper，SQL的面向流的扩展）。

从广义上讲，这类语言与SQL类似，并具有可读性，或者说这类语言与C语言类似，并具有相应的“神秘性”。作为两个极端的例子，让我们来看看StreamBase和KX。

5.3.1 StreamBase[①]

StreamBase直接对SQL的基本版本进行了扩展，SQL程序员能够轻易读懂。如预料的那样，关键字CREATE的作用是向模式（schema）中增加持久对象。下面是DDL语句的部分清单：

```
CREATE SCHEMA
CREATE TABLE
CREATE INPUT STREAM
CREATE OUTPUT STREAM
CREATE STREAM
CREATE ERROR INPUT STREAM
CREATE ERROR OUTPUT STREAM
```

注意，“流”被声明用于输入和输出，并进行错误处理。每个表都可以放在内存或磁盘。每个表都必须有一个主键，这是通过[USING{HASH|BTREE}]子句实现的。二级索引是可选的，通过CREATE INDEX语句声明。

数据类型更像Java而不是SQL。期望的类型有BOOL、BLOB、DOUBLE、INT、LONG、STRING和TIMESTAMP等。它们还包括结构化数据类型（LIST、TUPLE、BLOB）和应用于StreamBase引擎的特殊类型（CAPTURE、可命名的模式数据类型等）。TIMESTAMP是ANSI样式的日期和时间的数据类型，同时还可以被用作一个ANSI时间间隔数据类型。如同在SQL中一样，STRING不是固定长度的数据类型。BOOL类型可以是{TRUE，FALSE}。

对流进行扩展是合乎逻辑的。例如，我们可以使用一条语句来声明动态变量，从流中获取它们的值。想象一把深入“水流”中的勺子，语法如下：

```
DECLARE <variable_identifier> <data type> DEFAULT <default_value>
[UPDATE FROM '('stream_query')'];
```

① 声明：我做了StreamBase视频，可在http://www.streambase.com/webinars/real-time-data-management- for-data- professionals/#axzz2KWao5wxo观看。

在本例中，每次一个元组被提交给输入流 Dyn_In_Stream 时，动态变量 dynamic_var 就会改变其值。SELECT 子句用来组合输出流，动态变量被作为目标列表中的条目同时也在 WHERE 子句谓词中：

```
CREATE INPUT STREAM Dyn_In_Stream (current_value INT);
DECLARE dynamic_var INT DEFAULT 15 UPDATE FROM
 (SELECT current_value FROM Dyn_In_Stream);
CREATE INPUT STREAM Students
 (student_name STRING, student_address STRING, student_age INT);
CREATE OUTPUT STREAM Selected_Students AS
 SELECT *, dynamic_var AS minimum_age FROM Students WHERE age>=
 dynamic_val;
```

数据流的一个有趣的功能是节拍器和心跳。它是一个很好的比喻。下面是对应的语法：

```
CREATE METRONOME <metronome_identifier> (<field_identifier>, <interval>);
```

<field_identifier>是一个包含时间戳值的元组字段。<interval>是以秒为单位的整数值。METRONOME（节拍器）根据系统时钟定期提供输出元组。同样地，音乐家的节拍器可以用于指示一段音乐的确切节拍，METRONOME 可以用来控制下游操作的时序。

按照固定的时间间隔，METRONOME 会产生一个元组，其中包含带有名为<field_identifier>的一个时间戳字段，字段值是来自运行 StreamBase 应用程序的计算机的系统时钟的当前时间值。只要应用程序启动，METRONOME 就开始产生元组。

与其搭配的是 HEARTBEAT（心跳）语句，语法如下：

```
<stream expression>
 WITH HEARTBEAT ON <field_identifier>
 EVERY <interval> [SLACK<timeout>]
 INTO <stream_identifier>
 [ERROR INTO <stream_identifiers>]
```

<field_identifier>是输入流中保存时间戳数据类型的字段。<interval>是十进制秒值，表示其心跳发出元组的时间间隔。<timeout>是用来指定接收数据元组的允许延迟的十进制秒值。

HEARTBEAT 在与数据元组相同的流中加上定时器元组，以便下游操作能够在即便输入的数据中带有空白（或数据毛刺）也能执行。HEARTBEAT 会检测延迟或丢失的元组。像 METRONOME 一样，HEARTBEAT 使用系统时钟，并定期发出输出元组，但 HEARTBEAT 也能使用输入流中的信息产生元组，独立于系统时钟。

HEARTBEAT 会将输入元组直接传递到输出流，更新其内部时钟。如果一个预期的输入元

组在配置的<interval>加上<timeout>值的时间内仍然没有到达，那么 HEARTBEAT 就会使用除时间戳之外的所有 NULL 数据字段合成一个元组，并将其发出。

HEARTBEAT 位于流中，并毫无修改地传递数据元组。数据元组必须包含<timestamp>类型的字段。在可配置的时间间隔上，HEARTBEAT 会插入与数据元组具有相同模式的元组到流中。除<timestamp>字段外，源自 HEARTBEAT 的元组中的字段被设置为 NULL，<timestamp>字段总是具有有效值。

HEARTBEAT 直到第一个数据元组已经沿着流通过才开始发出元组。当输出<interval>恰好到达系统时钟或数据元组的<timestamp>字段达到输出<interval>的倍数的情况下，HEARTBEAT 就会发出元组。如果数据元组的<timestamp>字段的值大于即将到来的心跳 HEARTBEAT <interval>，HEARTBEAT 立刻发出目前流所需要的尽可能多的元组，以便使它的<timestamp>符合流中的当前<timestamp>值。

HEARTBEAT 产生一个流，并且可以在任何可以接受流表达的地方使用。下面的例子演示了如何使用 HEARTBEAT：

```
CREATE INPUT STREAM Input_Trades
(stock_symbol STRING,
 stock_price DOUBLE,
 trade_date TIMESTAMP);
CREATE OUTPUT STREAM Output_Trades_Trades;
CREATE ERROR Output_Trades STREAM Flameout;
```

这可以通过查询使用，像下面这样：

```
SELECT * FROM Input_Trades
 WITH HEARTBEAT ON trade_date
 EVERY 10.0 SLACK 0.5
 INTO Output_Trades
 ERROR INTO Flameout;
```

系统中与流协同工作的功能还有更多。流可通过 CREATE[MEMORY|DISK] MATERIALIZED WINDOW<window name>语句划分成有限个元组块，并且无限的流的这些子集可以像表一样被使用。就如一个“桶”从流中提取数据。BSORT 能够对流过缓冲区的稍微乱序的流数据（用户定义数据大小）进行重新排序。BSORT 产生一个新的、经过重新排序的数据流。还有其他工具，但这里作为概述已经够了。

虽然可以以文本文件的方式编写 StreamSQL，并编译它，但该公司还提供了一个图形工具 StreamBase Studio，让你可以通过画一个“水暖图”产生的代码。该图也可以作为不错的项目文档。

5.3.2　Kx[①]

Q 语言是由 Arthur Whitney 开发并且由 Kx 系统公司进行商业化的专有阵列处理语言。Kx 的产品在金融业已经使用超过 20 年。该语言作为 kdb+的查询语言，kdb+是基于磁盘/内存的列式数据库。

Q 语言 K 演变而来，K 由 APL 演变而来，APL 是“A Programming Language”的简称，由 Kenneth E. Iverson 和同事开发。它享有与 IBM 一样的时尚，例如，拥有特殊键盘，拥有神秘的符号，具有希腊符号、数学符号和其他特殊符号。

APL 语言的一个主要问题就是符号。IBM Selectric 打字键盘能容纳 88 个字符，APL 具有比它更多的符号，所以你不得不使用组合键。但是，Q 语言使用标准 ASCII 字符集。

这一系列语言使用了原子模型、列表和取自于 LISP 的部分函数。原子是不可分割的标量，包括数字、字符和时间数据类型。列表是有序的原子（或其他列表），在其上有更高级的数据结构，如字典和表，这些都由语言内部提供。语言中的运算符是由全部数据结构作为输入和输出的函数。这意味着，运算符没有优先级，代码由右至左执行，如同数学中的嵌套函数调用。想象一下，在数学中如何求 sin(cos(*x*))的值：先计算 *x* 的余弦，然后对其结果求正弦。括号会让代码变得杂乱，所以我们也使用数学符号 $f \circ g(x)$表示函数结构。在 Q 语言中，它们被简单地按顺序编写。

数据类型是我们所期望的类型（通过 SQL 等价性提供的），但时间类型比在其他语言中看到的更加完备。数值类型包括布尔（BIT）、字节（SMALLINT）、整数（INTEGER）、长整数（BIGINT）、实数（FLOAT）和浮点数（DOUBLE PRECISION）。字符串类型包括字符（CHAR(1)）和符号（VARCHAR(n)）。时间数据类型包括日期（DATE）、日期时间（TIMESTAMP）、分钟（INTERVAL）、秒（INTERVAL）和时间（TIME）。

该语言也使用 IEEE 浮点标准中的无穷大和 NaN 值。如果你不知道它们，就需要阅读一些资料。这些都是表示正无穷大和负无穷大的特殊值，以及那些符合它们所使用的特殊规则“非数字”的内容。

此外，还有一个 SQL 风格的 select [*p*] [by *p*] from texp [where *p*]表达式。该 by 子句在通常的 SQL 语法中已经发生了变化，最好用一个例子来展示。先看下面这个简单的表：

```
tdetails
eid | name iq
----| --------------
1001| Dent 98
```

① 声明：我为 Kx 制作了一个视频。

```
1002| Beeblebrox 42
1003| Prefect 126
```

以下语句及其结果如下所示。Count .. by ..语句对数据行中的 iq 字段进行排序并返回表中的相对位置。max 为 sc 在 from 子句表中的最高值：

```
select topsc:max sc, cnt:count sc by eid.name from tdetails where
 eid.name <> 'Prefect
name | topsc cnt
----------| ---------
Beeblebrox| 42 2
Dent | 98 2
```

千万不要被类 SQL 的语法迷惑。这仍然是一个列式数据模型，而 SQL 是面向行的。该语言包括列操作符。例如，从表中删除一列可以得到一个新的表结构。但这在 SQL 中是不会工作的。

同样地，你会看到过程式控制结构的实用版本。例如，在过程式语言中，赋值通常是用 SET、:=或=完成的，而 Q 语言使用冒号。显示在冒号左边的名称是冒号右边的表达式的名字；在过程式的意义上来讲，它并不是赋值。

如同 SQL 或 ADA 中的 CASE 表达式，Q 有下面的语句：

```
$[expr_cond1; expr_true1; . . . ; expr_condn; expr_truen; expr_false]
```

计数迭代通过经典的 do 循环的实用版本来实现：

```
do[expr_count; expr_1; . . . ; expr_n]
```

其中，expr_count 计算的结果值必须为整数。表达式 expr_1 到 expr_n 按照从左到右的顺序被求值 expr_count 次。do 语句并没有一个明确的结果，所以你不能在它里面嵌套其他表达式。下面是通过 n–1 次迭代形式的阶乘 n。循环控制参数为 f，参数为 n：

```
n:5
do[-1+f:r:n; r*:f-:1]
r
120
```

有条件的迭代通过经典的 while 循环的功能版本来实现：

```
while['expr_cond; expr_1; ... ; expr_n]
```

其中，expr_cond 经过运算后，只要 expr_cond 计算结果为“true”，表达式 expr_1 到 expr_n 就按从左到右的顺序求值。while 语句并没有一个明确的结果。

已经给了一些 Q 语言编码的体验，这一内容我不再多讲。它可能与你知道的语言有很大

的不同，这本书并不是一个 Q 语言的教程。你可以在 http://code.kx.com/wiki/JB:QforMortals2/contents 找到一个由 Jeffry A. Borror 编写的名为“Q for Mortals”的很棒的教程。

学习 Q 语言中要权衡的是，这些程序都运行得出奇地快并且很紧凑。有经验的 Q 程序员可以迅速通过编写代码来解决迫在眉睫的问题。

总结思考

复杂事件和数据流与这本书中的其他新技术类似。它们还没有成为大众熟知的、通用的、标准的语言。但是，我们生活在一个处理速度能达到光速世界里，真正的问题是在复杂事件真正发生前必须预先安排应对之策，事件发生之后或在事件期间是没有机会采取行动的。

参考文献

Chatfield, C. (2003). *The analysis of time series: Theory and practice*. Boca Raton: Chapman & Hall. ISBN: 978-1584883173.

Hohpe, G. (2005). *Your coffee shop doesn't use two-phase commit*. Available at http://www.enterpriseintegrationpatterns. com/docs/IEEE_Software_Design_2PC.pdf.

Brockwell, P. J., & Davis, R. A. (2009). *Time series: Theory and methods, Springer series in statistics*. New York: Springer_Verlag. ISBN: 978-1441903198.

Wald, A. (1973). *Sequential analysis*. Mineola, NY: Dover Publications. ISBN: 978-0486615790.

键值存储

简介

键值存储，也称为关联数组，从本质上来说是键值对（<键>，<值>）的集合，可以理解为一类简单的数组。在每个集合内部，键都是唯一的。键可以是任意可以检验相等性的数据类型。这是 MapReduce 系列的一种形式，但其性能取决于键的设计。因此，散列是很重要的技术。

这种模型只有下面 4 类基本操作：

- 向集合中插入键值对；
- 从集合中删除键值对；
- 更新存在的键值对的值；
- 查找与特定的键关联的值，未查到则返回异常。

6.1 模式与无模式

SQL 来源于 IBM 的一个叫做 SEQUEL 的项目，SEQUEL 代表“Structured English-like Query Language”的简写。这种结构来自 DDL，DDL 是定义了模式（schema）的 SQL 的子语言。文件与表完全不同，行也并不是记录，列也不是字段。行由列构成。列有已知约束的数据类型，并且是标量。列是无序的，但可以通过列名找到对应的列。

表是由行构成的。在表中，行是无序的。通过键来找到数据，而不是通过物理位置。SQL 中拥有表级约束和关系。最终，完整的模式就是工作单元，具有表与表之间间的约束和关系

(CREATE ASSERTION 和 FOREIGN KEY 及 PRIMARY KEY)。

空表也有结构。空表中的列依然是强类型的，其约束依然是强制的。对于空集来讲，所有的表级谓词都是适用的。而空文件就是什么也没有。例如，一卷空白的磁带就是空文件。但是，作为硬盘目录的条目，它拥有名字和 NIL 指针，或者其他文件结束标记作为它仅有的数据。因此所有的空文件看起来行为都是一致的。读空文件会立即得到一个文件结束标志。

如果没有模式，数据的完整性所需的约束就必须交给应用来处理。同样，表示层无法知道将会返回什么数据。这些系统会积极针对检索和增加操作进行优化，并且经常会提供除存储之外一些小功能。SQL 系统的安全性和强大的查询能力，被具有更好的可扩展性和性能的特定的数据模型所替代。

6.2　查询与检索

NOSQL 处理海量数据，而且不需要关系查询和完整性。数据可以是结构化的，但是这是额外的（非必须）。最典型的例子是拥有大量库存的在线商店，当客户访问网站时，我们想用 HTML 把一些图片和简短的文本描述显示在网页的一些位置上。本节剩余的部分将以网上鞋店作为例子进行说明。

NOSQL 的这种组织方式对于统计或元素不断增长的列表的实时分析特别有用，如 Twitter 的帖子或网络服务器日志，它们都是来自一个庞大的用户群。没有索引或约束检查，你需要处理海量的原始数据，而这些数据可能并不整洁。

6.3　“键”的处理

在这个环境中，键如何处理是至关重要的。必须针对目标数据设计键，而不是简单地将连续的数字作为键。对于鞋店，我们将有一些内部的产品标识符，但它可能会比较模糊，以至于当客户来访问网站的时候都不会注意到它。但是，如果我们能以用户能够明白的方式进行编码，大家使用起来也就更简单了。

好的键的设计并不是那么好找的。就像 Jorge Luis Borges 在他的论文“约翰·威金斯的分析语言”（The Analytical Language of John Wilkins）中做的阐述：

> 以下这些充满歧义、冗余和缺陷的文字来自 Franz Kuhn 博士的一本叫做《天朝仁学广览》的中国百科全书。在这本书里面，将动物分为（a）属皇帝所有的，（b）有芬芳香味的，（c）驯顺的，（d）乳猪，（e）鳗螈，（f）传说中的，（g）流浪狗，（h）包括在目前分类中的，（i）发疯似的烦躁不安的，（j）数目众多的，（k）浑身有十分精致的像骆驼毛刷的，（i）其他，（m）刚刚打破水罐的，（n）远看像苍蝇的。

对于网上鞋店来说，一个分众多种类的下拉列表并不会为搜寻鞋子带来多大的帮助。它没有明晰的键。许多年前，我曾为一家鞋公司设计了一个数据库。制造方这边需要基于鞋的物理性能报告，而营销方面更关心营销类策略。钢头工作靴属于制造类中的某一类。但在那个时候，市场上建筑工人和哥特风的女孩这两个截然不同的群体都会购买这类鞋子，而且他们也不在同一类店里消费。

理想的情况是通过一个键来迅速找到拥有所需的五金材质的实物商品，但在现实世界中，这几乎是不可能的，除非你的数据库处理的是有专门的分类系统的某一学科领域。

6.3.1 Berkeley DB

Berkeley DB（BDB）是一个软件库，也是应用最广泛的（<键>，<值>）数据库工具包。它能够被众多的软件使用是流行的一个重要原因。BDB 使用 C 写的，带有可以绑定到 C++、PHP、Java、C#、Perl、Python、Ruby、TCL、Smalltalk 和大多数其他编程语言的 API。每个（<键>，<值>）对可达 4 GB，而且每个键都可以有多个数据项。

BDB 在 20 世纪 80 年代末，在加州大学伯克利分校开始做的，直到 Oracle 在 2006 年 2 月收购了 BDB。结果导致它产生了一些合法的分支。虽然它是非关系型的，但是支持 ACID 事务。BDB 有一个加锁系统，可以利用锁机制来并发访问数据。有一个日志系统用来操作事务和数据恢复。Oracle 通过在 BDB 中扩展 SQLite 来支持 SQL。通过商业化产品 Metatranz StepSqlite 支持 PL/SQL 在 BDB 中的使用。

访问数据库的应用程序有权决定数据在记录中该如何存储，而且 BDB 不想对记录中的数据添加任何约束。我们回到文件模型，在文件模型中，是宿主程序给数据赋予意义。

6.3.2 通过树索引或散列访问

如果禁止使用常见的树索引和散列进行访问，（<键>，<值>）模型没有任何意义。最主要的区别是那些目标内容可能是在商业硬件的巨大“磁盘农场”中的其他硬件驱动器上。

6.4 “值”的处理

SQL 是强类型语言，它的每一列都有确定的数据类型、默认值和约束条件。我告诉人们，SQL 中实际工作的 85%～90%是在 DDL 中完成的。设计差的 DDL 将迫使程序员一遍又一遍地修正 DML。查询会重新检查谓词中的约束，这些约束是在 DDL 中只写过一次，然后被优化器从模式信息表中检索出来的。

（<键>，<值>）模型对值的本质并没有明确的规定。数据可以是结构化的，但是如果数据是结构化的，那么它的数据类型常常是简单、精确的，并且可以是数值类型、字符串类型、时

间类型等，并且没有默认值和约束条件。因为这种模型的目的是能够存储和检索大量数据，而不是元素之间的关系。在老式 FORTRAN 文件系统中也存在所谓的记录，但不是 RDBMS 中的行。

优点是任何主机语言程序都可以立即使用数据。但是，如果你看一下 SQL 标准，每种 ANSI X3J 语言都有将其数据类型转换为 SQL 数据类型的规则。还有一些指明了如何处理空值和在主机与数据库之间传递其他信息。

6.4.1　任意字节数组

非结构化数据也保留在这些产品中。通常的格式是某种字节数组：BLOB、CLOB 或者其他大“块”的连续原始数据。主机程序决定数据的意义以及如何使用它。（<键>，<值>）存储希望能够尽可能快地移动数据到主机上。

移动数据的最快方法是把它在主存上而不是辅存上。注意，我用的“主”而不是“核心”“RAM”等，是“辅存”而不是“磁盘”“DRUM”“RAID 等。这种技术进步很快，主存（快速，直接寻址的处理器）和辅存（慢，需检索其磁盘）之间的界线越来越模糊。我相信，2015 之后的几年内，SSD（固态硬盘）将会取代移动硬盘。这将彻底改变我们看待计算的方式。

- 主存和辅存速度没有差异。
- 内核（之前所谓的 CPU）也很便宜，我们将把它们放在各级硬件上。我们将掌控并行编程。
- 结果将是，并发表扫描的速度会比用索引速度快，并且与散列的结果没有什么不同。

总之，最终都会成为一个内存数据库。

2013 年 4 月，IBM 发布新闻称硬盘驱动器将很快在企业中消失。他们花费数 10 亿美元来支持这一预测。IBM 已经推出了一系列基于闪存的存储系统，称为 FlashSystems。FlashSystems 源于 IBM 在 2012 年收购的 Texas Memory。一套 FlashSystems 可通过配置，使单个机架上的数据存储达到 1 PB，并可生产 2200 万 IOPS（每秒 I/O 操作数）。而从硬盘系统中获得同样级别的存储和吞吐量需要配备 315 台拥有高性能磁盘的机架。

IBM 也有闪存和磁盘的混合存储系统，包括 IBM Storwize V7000、IBM System Storage DS8870 和 IBM XIV Storage System。他们认识到，当前并不是所有系统都会受益于固态技术的使用。性能必须是系统运行的关键因素。2013 年，普通硬盘的成本约为 2 美元/GB，一个企业的硬盘成本约为 4 美元/GB，一个高性能的硬盘驱动器的成本约为 6 美元/GB。固态硬盘是 10 美元/GB。

6.4.2　已知结构的小文件

由于（<键>，<值>）存储可用于网站，一种值就是保存目录页或提供的产品的小文件。

XML 和 HTML 是非常理想的。每种浏览器都可以使用它们——超文本链接是很容易的，它们是简单的文字，很容易更新。

6.5 产品

下面是摘自维基百科的 2013 年产品的快速清单，并且采用了维基百科的分类。

最终一致性（<键>，<值>）存储：

- Apache Cassandra
- Dynamo
- Hibari
- OpenLink Virtuoso
- Project Voldemort
- Riak

分层（<键>，<值>）存储：

- GT.M
- InterSystems Caché

云或主机服务：

- Freebase
- OpenLink Virtuoso
- Datastore on Google Appengine
- Amazon DynamoDB
- Cloudant Data Layer（CouchDB）

RAM 中的（<键>，<值>）缓存：

- Memcached
- OpenLink Virtuoso
- Oracle Coherence
- Redis

- Nanolat Database
- Hazelcast
- Tuple space
- Velocity
- IBM WebSphere eXtreme Scale
- JBoss Infinispan

固态盘或旋转盘（<键>，<值>）存储：

- Aerospike
- BigTable
- CDB
- Couchbase Server
- Keyspace
- LevelDB
- MemcacheDB（使用 BDB）
- MongoDB
- OpenLink Virtuoso
- Tarantool
- Tokyo Cabinet
- Tuple space
- Oracle NoSQL Database

有序（<键>，<值>）存储：

- Berkeley DB
- IBM Informix C-ISAM
- InfinityDB
- MemcacheDB
- NDBM

总结思考

因为这是用于数据检索的最简单和最普通的模型之一，有大量使用这种模型的产品。使用这种技术的真正的技巧是键的设计。

文本数据库

简介

我十年前就创造了文本库（textbase）这个词，用来定义才刚刚开始的演变。最重要的业务数据并不是存在数据库或文件中，而是存在文本中。这些数据可能是合同/契约、担保条款、信件、手册和参考资料。传统意义上来说，数据处理（另一个旧术语）是指处理在机器可用介质中的高度结构化的数据。存储是昂贵的，你不能用文本把空间浪费掉。文本天然就是模糊并且笨重的，传统的数据需要被精确和紧凑地编码。

文本和印刷打印的文档也有法律上的问题。由于已经被发明了几千年，它们有法律和社会机制强制的要求和传统。随着存储变得越来越便宜，人们变得越来越聪明，文字印刷开始实现自动化。不只是文字印刷，阅读和理解这些文字也变得自动化。

7.1 经典文档管理系统

文本数据库开始作为文档管理系统。早期的是缩微胶卷和缩微胶片。文本被存储为与机器可搜索的索引相关联的物理图像。1938 年，UMI（University Microfilms International）由 Eugene Power 创建，发行当前和过去的出版物以及学术论文的缩微胶片版本。他们一直主宰这个领域到 20 世纪 70 年代。

最大的变化是法律上的，而不是技术上的。首先，FAX 的文档复制成为合法的，然后被签名的缩微胶片拷贝成为合法的，最后电子签名的签名复制成为合法的。电子签名的法律定义可以是一个简单的复选框，也可以是签名的图片文件，还可以是一种可以由第三方验证的

加密协议。但最终的结果是，你再也不需要一个装满了纸质的法律文件的仓库。

7.1.1　文件索引和存储

缩微胶卷的帧之间有物理“光点”，以便缩微胶片的读者可以对帧进行计数和定位。缩微胶片设备在放有胶卷的窗口上使用霍尔瑞斯卡（是的，文献中常常会说成“霍尔瑞斯”[①]）。可以用打孔卡分拣机和存储柜处理它们。这些都是先前的磁带和打孔卡逻辑模型！这没什么好惊讶的，是新技术模仿了旧技术。人们不会立即就跳转到新的思维方式。

后来，我们有了可以物理移动缩微胶卷或缩微胶片的硬件。它们通过各种机械零件进行旋转。如果你能找到以前关于这些机器的视频，你会对什么是“蒸汽朋克[②]”（steam punk）有一些感觉，并且怀疑这个科幻风格是不是在 20 世纪 50 年代，而不是维多利亚时代。我们称为“机电朋克”（electromechanical punk），大家会穿灰色法兰绒西装，戴窄领带。

这些是 NoSQL 所使用的（<键>，<值>）模型的一个版本，它使用更多的物理技术。但是是有区别的。在文本数据库中，最终判断是根据人对文档的阅读和理解做出的。针对半结构化文档，如保险单，会带有策略号码以及其他传统的结构化数据元素。但也有半结构化数据元素，如体检。而且有一些完全免责的文本元素，例如，像这样一条笔记：“调查这家伙是否存在诈骗行为！我们认为他为了骗取保险杀害了他的妻子！”或者更糟。

7.1.2　关键字和题内关键字

如何帮助人们更好地理解文档？一个人会首先接触一个文档的标题。这听起来很明显，这就是为什么杂志和日常的文档虽内容平淡却有极具描述性的标题。20 世纪初以前，书还会有一个副标题，以帮助别人决定是否购买这本书。青少年小说会尤其遵守这种方式，成年人小说以及学术著作也是如此。

进化过程中的下一个阶段是从一个专业词汇表选出了关键字的列表。这在技术杂志中很流行。明显的问题是选择术语列表。在一个技术领域中，只是紧跟术语是很困难的。备用拼写、缩写词和替换术语等情况任何时候都会发生。就像医生在病例中记录病人的疾病时随着时间推移疾病的名字会有所改变一样。

由于真正的数据是语义，进化的下一阶段是题内关键字（keyword in context，KWIC）索引技术。它是在文件中词汇索引线中最常见的格式。词汇索引就是文档中所用的主要单词按照字母顺序排列的列表。在计算机诞生前，这些工作是困难、耗时且费用昂贵的。这就是只会在宗教经文或主要的文学作品中使用词汇索引的原因。第一本书籍的词汇索引，是在拉丁

① 利用凿孔把字母信息在卡片上编码的一种方式，以美国发明人赫尔曼·霍尔瑞斯（Herman Hollerith）的姓氏命名。——译者注

② 参考 https://zh.wikipedia.org/wiki/蒸汽朋克。——译者注

文圣经上完成的，是由修·圣雪儿（卒于 1262 年）雇用 500 僧侣协助他进行编纂。1448 年，犹太教教士 Rabbi Mordecai Nathan 完成了希伯来圣经的词汇索引，他花了 10 年时间。今天，我们可以使用具有更好的响应时间的文本搜索程序来协助完成。

KWIC 系统及其衍生系统都基于一个称为“标题内关键词”的概念，这个概念在 1864 年第一次由曼彻斯特图书馆的安德烈·斯塔多罗（Andrea Crestadoro）提出。KWIC 索引通过很明显的分割线将每行分割成垂直的两列，关键词在分割线的右侧，按照字母顺序排列。为了减少排版，分割线就是在文本内容中的垂直空白。

KWIC 也有一些变种，如 KWOC（keyword out of context，题外关键词）、KWAC（keyword augmented in context，题内增强关键字）和 KEYTALPHA（key term alphabetical，字母顺序关键词）。这些的差异是它们以不同的形式显示给用户。关键字索引方法好的一面是，一旦有一个关键字列表，关键字索引很快建立。它就像为传统的文件或数据库做索引一样简单。许多系统甚至不需要控制词汇表，只依赖控制扫描就能构建这个索引列表。

坏消息（进化的另一阶段）是，搜索中可能没有任何语义。相关主题的概念，或更窄或更宽的那些可被识别分级结构的概念和主题，在用户的脑海外并不存在。

7.1.3 行业标准

严肃的文档处理始于图书管理员，而不与计算机用户。这毫不奇怪，早在数据库之前书籍就已经存在。美国国家信息标准化组织（National Information Standards Organization，NISO）及现在的 ANSI Z39 小组，成立于 1939 年，这是在计算机普及很久以前。它是出版业、图书馆、IT 和传媒组织等领域中 70 多个组织的保护伞。他们在很多领域有很多的标准，但对我们重要的一条是处理文档，而不是处理图书馆书架。

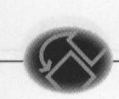
1. 上下文查询语言

NISO 定义了最小的文本语言，通用查询语言（Common Query Language，CQL）（http://zing.z3950.org/ cql/intro.html）。它假设有一组文件具有可查询的计算机化的接口。查询可以使用 3 种常见的布尔运算符（AND、OR 和 NOT）在文档中搜索词或短语（字符串用双引号）。例如：

```
dinosaur NOT reptile
(bird OR dinosaur) AND (feathers OR scales)
"feathered dinosaur" AND (yixian OR jehol)
```

所有的布尔运算符都具有相同的优先级，并由左到右进行结合。这是不符合程序员预期的！这意味着需要使用大量额外的括号。下面这些都是相同的查询：

```
dinosaur AND bird OR dinobird
(dinosaur AND bird) OR dinobird
```



邻近运算符基于文档中的词之间的位置关系选择候选者。下面是中缀运算符的 BNF（巴克斯范式或巴科斯范式，一种正式的编程语言文法）：

```
PROX/[<relation>] / [<distance>] / [<unit>] / [<ordering>]
```

然而，如果仅需要默认值，任何一部分参数或全部参数都可以省略。此外，由斜杠组成的操作符的尾部（因为使用默认值）整个都被省略。这很容易用下面的例子来解释。

```
foo PROX bar
```

这里，单词 foo 和 bar 立即彼此相邻，顺序任意。

```
foo PROX///SENTENCE bar
```

单词 foo 和 bar 出现在同一个句子的任何地方。这意味着该文档引擎可以检测句子，这是关于语法和语义的边缘。当单位不是单词时，默认的距离为零。

```
foo PROX//3/ELEMENT bar
```

单词 foo 和 bar 必须在距离对方 3 个元素内出现。例如，如果一个包含作者列表的记录，第 4 个作者的名字中包含 foo 且第 7 个作者的名字中包含 bar，那么这个搜索就会发现该记录。

```
foo PROX/=/2/PARAGRAPH bar
```

在这里，单词 foo 和 bar 必须严格地分开出现在两段中，即便它们出现在同一段落或相邻段落中，也是不满足条件的。而我们现在用“段落”作为数据搜索的单位：

```
foo PROX/>/4/WORD/ORDERED bar
```

这条搜索语句会查找出现该单词的记录，以 foo 为第一个单词，随后 4 个以上的单词后跟随 bar，并且是按照这样的顺序。其他搜索是不排序的。

一份文档中可以有可被搜索的索引字段（年、作者、ISBN、书名、主题等）。同样，下面的例子会给你一个概述：

```
YEAR>1998
TITLE ALL "complete dinosaur"
TITLE ANY "dinosaur bird reptile"
TITLE EXACT "the complete dinosaur"
```

ALL 选项指明该搜索操作会查找字符串中所有的单词，不考虑它们的顺序。ANY 选项指明该搜索操作会查找字符串中的任何单词。EXACT 选项指明该搜索操作会查找字符串中所有的单词，并且它们必须要按照给定的顺序。精确搜索对结构化字段（如 ISBN 号、电话号码等）是最有用的。但是，下面我们将了解修饰符语义的内容。通常反斜杠修改符会用于：

- STEM：作为搜索条件的单词会被扫描并匹配从相同根词派生的词。例如，wallked、walking、walker 等在搜索中都将简化到根词——walk。显然，这是依赖于语言的。
- RELEVANT：作为搜索条件的单词与被搜索的记录存在一定的依赖关系。例如，搜索主题 ANY/RELEVANT "fish frog"能够查找出在主题字段中包含 shark、tunacoelocanth、toad、amphibian 等任何单词的记录。
- FUZZY：一个包罗万象的修饰符，表示服务器可以在指定的搜索词及其记录中应用“模糊匹配”的实现依赖形式。这可能对拼写错误的搜索字词非常有用。
- PHONETIC：这个修饰符不仅是尝试匹配那些拼写相同的单词，还会匹配那些发音相同的词。例如，SUBJECT =/PHONETIC，rose 可能匹配 rows 和 roes（鱼卵）。

修饰符 EXACT/FUZZY 看似很奇怪，但对于有错误的结构化字段非常有用。例如，对于可能有不正确位数的电话号码来讲，结构是精确的那部分，但位数是模糊的那部分：

```
telephone_nbr EXACT/FUZZY "404-845-7777"
```

2. 商业服务和产品

已经有许多商业服务，提供对数据的访问支持。使用最多的是 LexisNexis 公司和 WestLaw 的法律研究服务。他们有 TB 级的在线数据通过订阅的方式在网上销售。也有其他行业专用的数据库服务、以大学为基础的文档文本数据库等。

这些服务具有专有的搜索语言，但大多数这些语言都与 CQL 类似，只是在语法上有细微的差别。它们具有布尔型和邻近构造，但通常包括对特定专业的支持，如法律或医学词汇。

后来，还有产品可以基于用户的原始数据创建文本数据库。ZyIndex 在这一领域领先了几十年。它是写于 1983 年为运行 DOS 的 IBM 兼容计算机上的文件用 Pascal 语言编写的全文搜索程序。多年来，这家公司新增了光学字符识别（OCR）、全文搜索电子邮件及附件、XML 和其他功能。随后的竞争以及内部就够都发生了变化，但大多数的搜索语言都恪守 CQL 模型。

这些产品的实现分为两个阶段。第一阶段是建立文档的索引，以便它们的内容可以被搜索。建立索引同样意味着原来的文本可以被压缩，这很重要！第二阶段是搜索引擎。建立索引要么忽略“干扰词”要么使用全文内容。干扰词的概念是语言上的，在中国汉语语法中的“全词”和“空词”。干扰词或空词就像连词、介词、冠词等结构性的东西，而不是构成语言骨架的动词和名词。事实上，每一句话都会有干扰词，因此找出它们是一种浪费。值得权衡的是“做还是不做”或是其他词语都是干扰词，但可能不会被发现。

有一些系统在对内容建立索引时将扫描每一个词。大多数模型是与数据页号的列表而不是其在文件中的精确位置进行关联的。按照数据页的方式抓取文件的内容是很容易的，因为磁盘存储也是这样的工作原理。一旦它（搜索的内容）在主存储中，就可以简单、低成本地

进行解压缩并显示文本。

3．正则表达式

正则表达式来自 UNIX，由数学家 Stephen Cole Kleene 创造。它是用以描述一组字符串的抽象模式字符串。ANSI/ISO 标准 SQL 有一个简单的 LIKE 谓词和更复杂的 SIMILAR TO 谓语。

竖线分隔替代：gray|grey 能够匹配 gray 或 grey 作为一个字符串。括号用于定义操作的范围和操作符的优先级（参考其他用法）。记号（如一个字符）或组之后的量词指定了前面的元素允许出现的次数。最常见的量词是问号（?）、星号（*）（派生自 Kleene 星号[①]）和加号（+）。在英语中，它们表示的含义如下。

- ?：匹配前面的子表达式零次或一次。例如，colou?r 同时匹配“color”和“colour”。
- *：匹配前面的子表达式零次或多次。例如，ab*c 可以匹配“ac”“abc”“abbc”“abbbc”等。
- +：匹配前面的子表达式一次或多次。例如，ab+c 匹配“abc”“abbc”“abbbc”等。

这些结构可以结合使用，形成任意复杂的表达式。不同工具间正则表达式的实际语法各不相同。正则表达式是纯粹的语法，没有任何语义。SQL 工程师喜欢这一点，但文本数据库工程师是按照语义进行思考的，而不是语法。文本数据库不完全使用正则表达式的根本原因就在这里。

IEEE POSIX 基本正则表达式（Basic Regular Expressions，BRE）标准（ISO/IEC 9945-2:1993）的设计主要是为了与传统的（简单正则表达式）语法向后兼容，但又对许多 UNIX 正则表达式工具中已被采用为默认的语法提供了一个通用的标准，尽管经常有一些变化或附加功能。在 BRE 语法中，大多数字符唯一匹配的字符是它自己（例如，“a”匹配“a”）。但有一些例外情况，如表 7-1 列出的，它们称为元字符或元序列。

表 7-1　BRE 异常

元字符	描述
.	匹配任何单个字符（许多应用程序排除换行，至于哪个字符被认为是换行是字符编码和平台特有的，但它假设包含换行符是安全的）。在 POSIX 的方括号表达式（[]）中，点字符（.）匹配字面含义的点（.）。例如，a.c 匹配“abc”，但[a.c]只匹配“a”“.”或“c”
[]	匹配包含在括号中的任意一个字符。例如，[abc]匹配“a”“b”或“c”。[a-z]指定了一个范围，匹配“a”到“z”的任何小写字母。这些形式可以混合：[abcx-z]匹配“a”“b”“c”“x”“y”或“z”，其含义与[a-cx-z]相同

① 参见 http://zh.wikipedia.org/wiki/克莱尼星号。——译者注

续表

元字符	描述
[^]	匹配不包含括号内字符的单个字符。例如，[^abc]匹配除“a”“b”“c”之外的任何字符。[^a-z]匹配任何不是从“a”到“z”小写字母的单个字符。同样，字面量字符和范围可以混用
^	匹配字符串中的起始位置。在基于行的工具中，它与任意行的起点位置相匹配
$	匹配一个字符串结尾的位置或是一个字符串结尾后换行之前的位置。在基于行的工具中，它匹配任意行的结尾的位置
\n	匹配第 *n* 个标记的子表达式匹配的内容，其中 *n* 是从 1 到 9 的数字。这种构造在理论上是不规范的，在 POSIX ERE 语法中未获通过。有一些工具允许引用超过 9 个捕获组
*	匹配前面出现的元素零次或多次。例如，ab*c 匹配“abc”“abbbc”等，[xyz]*匹配“,”“x”“y”“z”“zx”“zyx”“xyzzy”等，(ab)*匹配“ab”“abab”“ababab”等

7.2 文本挖掘与理解

到目前为止，我们所讨论的都是使用本地文本数据库做检索的简单工具。当然，文本的检索是非常重要的，这并不是我们处理文字的真正的目的。我们需要的是文字的意思，而不是字符串本身。这就是为什么第一个被律师以及其他专业人才使用的工具使用了专业的词汇，并从专有的来源中获取文本。

如果你想折腾较大文本数据库，Google 提供了一个在线工具（http://books.google.com/ngrams），可以对 1800 年至 2009 年之间的许多语种的图书搜索短语出现的比率做成一个简单图表。这可以让研究人员找出一个名词的传播过程，它在何时流行，以及在何时过气。

我们必须摆脱这种有限的范围，并且需要搜索一切内容，为此，产生了 Google、雅虎等网络搜索引擎。文字的数量和它的不断变化显然是一个问题，更微妙的问题是文本的混合的性质，对大量的文字即使做简单的搜索也是非常费时的。

Google 发现，有多少人搜索流感相关的主题，与有多少人实际上有流感症状这两者间存在很强的相关性。当然，不是每个搜索“流感”的人实际上真生病了。但是，当所有与流感相关的搜索查询都出现时，这个模式就会出现。有一篇文章（http://www.nature.com/nature/journal/v457/n7232/full/nature07634.html）中给出了一些细节。

7.2.1 语义与语法

字符串是简单的模式。你不介意用于构建字符串的字母是拉丁文、希腊文，还是你发明的符号。一个字符串的最重要的属性是字母的线性排序。机器喜欢线性排序，它们善于在给

出了一组语法规则后进行解析。我们有坚实的数学基础以及大量的软件进行分析。处理字符串的生活是美好的。

单词是不同的，它们具有意义。单词构成句子，句子构成段落，段落构成完成的文档。这就是语义。这是阅读理解。这是计算机科学中人工智能技术（AI）的目标之一。计算机极客有一则笑话：计算机的研究中一切已经运行的都是AI，但这并不是全部。

读取学生论文和对学生论文进行评级的程序称为robo读卡器（robo-reader），它们是有争议的。在2013年3月13日，麻省理工学院的前写作主任Les Perelman在内华达州拉斯维加斯的大学生构成与沟通大会上提交了一篇论文。这是他对阿克伦大学教育学院院长Mark D. Shermis的2012年的论文的批评。Shermis和合著者Ben Hamner（数据科学家）发现，在分级SAT作文中，自动作文评分有能力产生与人工评分相似的分数。

Perelman认为方法论和Shermis-Hamner论文中的数据并不能支撑他们的结论。之所以计算机评级重要是因为大多数州（美国的行政区域）正计划推出新的针对K-12学生的高风险测试，这种测试需要写作部分。现在市面上销售教育软件，是因为根本没有足够的人手做相应的工作。

文章评级软件是由一些主要供应商生产的，其中包括McGraw-Hill和Pearson。有两个州联盟共同准备在2014年针对通用核心课程推出全新的高风险的标准化考试，同已经风靡全美的方式相匹配。两大联盟——Partnership for Assessment of Readiness for College and Careers和Smarter Balanced Assessment Consortium，希望利用机器来降低成本。

Perelman认为，教师很快就会教学生如何写作，以取悦机器人读卡器（以便获得高分），Perelman认为机器人会根据文章的长度和措辞不成比例地给学生分数，即使这些文章内容并不十分有意义。他指出，“这台机器会被设计为尽可能接近人评估的分数，但机器并没有办法理解文章的含义”。在2010年，24个成员州都制定了通用核心考试，该小组希望使用机器为所有的写作进行评级。今天，这种情况已经改变，现在写作部分占40%，40%为在阅读文章然后写入答案，并在数学部分有25%书面答复是人工评分的。

许多年前，*MAD*杂志发表了一篇幽默的文章，这篇文章是关于高中教师评级著名演讲和文学作品的。亚伯拉罕·林肯曾被告知，由于表达清晰度的问题，要用“87年”取代“4个20年又7年”；欧内斯特·海明威需要为自己的文章添加更多的描述；莎士比亚也曾被告知“生存还是毁灭”是矛盾的，我们需要对这个问题给出一个明确的说法。“我们认为，技术进步的速度并没有我们思考问题的速度那样快。”智慧平衡评估协会主管杰奎琳·金说。我也并不确定，技术真的可以做到遥遥领先。

7.2.2　语义网

语义网络（semantic network）或语义网（semantic web）是一个关于语言的图模型（见第

3 章）。粗略地说，文法用图的弧和作为节点的单词来展示。节点需要一个释义的过程，引用 Led Zeppelin 的《天国的阶梯》（*Stairway to Heaven*，1971）的歌词来说，这是“因为你知道，单词有时候有两层含义”，软件需要学习如何挑选一个意思。

字义消歧（Word-sense disambiguation，WSD）是自然语言处理的一个未解的问题，有两个变种：词法样本和所有词。词法样本使用的预选词的小样本。所有单词方法都使用来自 UNIX 的数学家 Stephen Cole Kleene 的正则表达式。正则表达式是一组描述字符串的抽象模式字符串。ANSI/ISO 标准 SQL 有一个简单的 LIKE 谓词和更复杂的 `SIMILAR TO` 谓词。这是更实际的评估形式，但它的成本比较昂贵。

考虑（书写）单词“bass”：

- 一种鱼（fish）；
- 低频音调。

如果你听到别人说这个词，可能会感觉有发音的差异，但这些句子没有给出这样的线索：

- 我去钓鲈鱼（I went bass fishing）；
- 这首歌的低音太弱（The bass line of the song is too weak）。

你需要一个上下文。如果第一句出现在文本中，解析可以看出，这可能是一个形容词-名词模式。但是，如果它出现在 *Field & Stream* 杂志[①]的一篇文章中，在语义上你就会更有信心——“bass”的意思是一条鱼。但是，如果它出现在年度 *Downbeat*[②]音乐评论中，其语义则可能是“低频音调”。

WordNet 是一个把英语单词放到成组的同义词（叫做同义词集）的英语词汇数据库。还有简短的、通用的定义以及同义词集之间的各种语义关系。例如，“car”的概念被编码为{“car”, “auto”, “automobile”, “machine”, “motorcar”}。WordNet 是一个被广泛使用的开源数据库。

7.3 语言问题

美国人往往不学外语。在文化方面，我们相信每个人都学美式英语，今天，这种情况在很大程度上是真实的。我们继承了旧的大英帝国的遗产，并加入美国的技术和经济来管理它。当计算机出现后，我们的技术发展带给了我们另外一个优势。我们让基本的拉丁字母能够嵌入 ASCII，所以很容易将英文文本存储在计算机上。字母是线性的并且具有小的字符集，其他语言则没有这么幸运。中文需要一个巨大的字符集，韩文及其他语言则要求字符写在两个维度。

① http://en.wikipedia.org/wiki/Field_%26_Stream，一本关于狩猎、钓鱼内容的杂志。——译者注

② http://zh.wikipedia.org/wiki/Down_Beat，一本关于爵士、蓝调音乐的美国杂志。——译者注

7.3.1　Unicode 和 ISO 标准

这个世界上存在更多的字母、音节和符号系统，不仅仅是 Latin-1。今天，又有了 Unicode。这是一种 16 位编码，它可以表示世界上大多数的文字系统。它拥有超过 11 万个字符，覆盖 100 种文字。该标准给出了每个字符的打印版本、规范化规则（如何从重音标记和其他叠加层组装字符）、分解、归类和显示顺序（从右到左的脚本与左到右的脚本）。

Unicode 是用于计算机软件国际化和本地化的事实上的和法律上的工具。它是 XML、Java 和.NET 框架的一部分。在 Unicode 之前，ISO 8859 标准中为大部分欧洲语言定义了 8 位代码，基本上能够完全或部分覆盖。问题是，任意文字彼此混合的文本数据库会如果没有严谨的编程实无法存储的。Unicode 的前 256 码点与 ISO8859-1 是相同的。这让我们不仅能够将现有西方语言文本转换为 Unicode，还能够存储 ISO 标准编码，当然，仅限于字符的这个子集。

Unicode 为每个字符提供了一个唯一的数值码点。它不处理显示，所以 SQL 程序员会更喜欢这样的抽象方式。表示层必须决定大小、形状、字体、颜色或任何其他可视属性。

7.3.2　机器翻译

在语言的文法中存在屈折[①]和黏着[②]部分。屈折语言改变了语言的形式来显示文法功能。在语言学中，变格[③]是名词、代词、形容词和用来指示数目的冠词（至少分单数和复数，阿拉伯语中还具有双数）、案例（主格的、主观的、属格、所有格、工具格、区位等）以及性别等词语类型的屈折语为了实现特定的语法功能而产生的词形变化。古代英语是一种高度屈折语言，但随着它演变成现代英语其词形也大大简化了。

表 7-2 中给出了一个拉丁名词变格的例子，使用单词 homo 的单数形式。

表 7-2　homo 的拉丁变格

单词	变格
homo	(nominative) subject
hominis	(genitive) possession
hominī	(dative) indirect object
hominem	(accusative) direct object

① 参见 http://zh.wikipedia.org/wiki/屈折语。——译者注
② 参见 http://zh.wikipedia.org/wiki/黏着语。——译者注
③ 参见 http://zh.wikipedia.org/wiki/变格。——译者注

续表

单词	变格
homine	(ablative) various uses
—	(vocative) addressing a person
—	(locative) rare for places only

还有更多变格，拉丁语还不是这类语言中最糟糕。这些最糟糕的语言难以解析并且其语法往往是高度不规则的。它们对于相关的概念往往有完全不同的词，如复数形式。英语因为其复数（mouse 和 mice、hero 和 heroes、cherry 和 cherries 等）而臭名昭著。

在另一种极端情况下，黏着语通过前或后固定词素与词根或词干形成新单词。相对于拉丁语，最典型的粘着语言是世界语（Esperanto）。这种语言的优点是解析起来要容易得多。例如：

“I see two female kittens”——Mi vidas du katinidojn

其中 kat/in/id/o/j/n 意思是“female kittens in accusative”，并且由下列元素组成：

kat (cat=root word) || *in (female gender affix)* || *id (“a child” affix)* || *o (noun affix)* || *j (plural affix)* || *n (accusative case affix)*

词根和词缀不会改变。这就是为什么像分布式语言翻译（Distributed Language Translation，DLT）这样的项目选择使用某个版本的世界语作为中间语言的原因。所述文本数据库被翻译成中间语言，然后从中间语言转换成目标语言之一。世界语非常简单，作为中间语言可以在目标计算机上被翻译。DLT 在欧洲被用于有线电视滚动视频新闻文本。

总结思考

我们正在用文字做很多事情，我们从来没有想过这些事情会是可能的。但机器不能根据直觉做出跳跃，它们只能用在很窄的专业领域。

2011 年，IBM 把他们的沃森人工智能（Watson AI）系统推上了电视智力竞赛节目《危险边缘》。程序必须回答用自然语言提出的问题。它非常令人印象深刻，赢得了 100 万美元奖金。这台机器是专门为回答该节目上的问题而开发的，它并不是通用工具。使用包含只有几个单词的线索的问题类别是有问题的。2013 年 2 月，沃森的第一个商业应用是在纪念斯隆-凯特琳癌症中心的健康保险公司 WellPoint。现在它被调整到在肺癌治疗中利用管理决策。

凭借改进的算法和计算能力，我们能够找到阅读和理解文字的要点。这是一个完全不同的游戏，但它还不是人类智能。这些算法找出存在的关系，不创造新关系。我最喜欢的例子

是医学中烧伤外科医生在他烧烤时看到可以扩大的金属网格，意识到相同的模式（狭缝和伸展）可能可以用于修复人体皮肤的烧伤。这种思考机器是做不到的。

LexisNexis 公司和 WESTLAW 教法律系学生使用他们的文本数据库进行法律研究的方法。这些课程包括基于著名的和重要的法律判决的标准练习。年复一年，法律系的学生绝大多数都无法完成一个正确的搜索。他们常常得到自己并不需要的文档，却无法找到自己应该找到的文档。即使是从事特殊工作的聪明的人也并不善于从文本数据库中检索问题。

参考文献

Perelman, L. C. (2012). *Critique (Ver. 3.4) of Mark D. Shermis & Ben Hammer, "Contrasting State- of-the-Art automated scoring of essays: Analysis"*. http://www.scoreright.org/NCME_2012_Paper3_29_12.pdf.

Ry Rivard, Ty (2013, Mar 13). *Humans fight over Robo-Readers*. http://www.insidehighered.com/news/2013/ 03/15/professors-odds-machine-graded-essays#ixzz2cnutkboG.

Shermis, M. D., & Hamner, B. (2012). *Contrasting State-of-the-Art automated scoring of essays: Analysis. The University of Akron*. http://www.scoreright.org/NCME_2012_Paper3_29_12.pdf.

地图数据

简介

地理信息系统（Geographic information system，GIS）是一个存储地理、地理空间或者时空数据的数据库。它不仅是一种地图制图。我们不仅是试图在地图上定位某样东西，还试图找到数量、密度、一个地区的内容、在一段时间中的改变等。

这种类型的数据库可以追溯到1832年法国地理学家Charles Picquet为早期流行病所做的研究准备“Rapport sur La Marche Et Les Effets du Choléra Dans Paris et Le Département de la Seine”。他用颜色将巴黎的地图划分为不同的区域，以便显示该地区每千人有多少人死于霍乱。

1854年，John Snow制作了一张类似的伦敦霍乱爆发的地图，并且用点去标注一些个体病例。这项研究是统计学和流行病学的一个经典案例。伦敦使用公共的手摇水泵，市民用桶运水到自己的住宅。他的地图清晰地展示了病源来自于Broad街的水窖，这里的水已经被污水污染。霍乱开始就伴随着腹泻，腹泻使人体一天损失多达20升水，导致70%的患者迅速脱水死亡。几天之后，死亡率急剧下降，几乎接近于0。我们拥有制图技术已经几个世纪了，但是John Snow地图是首例分析一系列地理相关现象的。

请记住这些图都是手绘的。直到20世纪早期我们才拥有摄影技术，20世纪晚期才拥有计算机学。第一个真正意义上的GIS是加拿大的CGIS，由加拿大林业和农业开发部开发。主导这个项目的Tomlinson博士也被称为“GIS之父”，不只是因为GIS这个术语，而是因为他首次在应用中覆盖了对收敛地理数据的空间分析。在这之前，计算机地图绘制技术只是用于地图而没有应用数据分析。

可惜的是，CGIS 从未成为一个商业产品。不过，它却激励了跟随它的所有商业产品。当今，最大的玩家是环境系统研究所（Environmental Systems Research Institute，ESRI）、计算机辅助资源信息系统（Computer-Aided Resource Information System，CARIS）、地图显示和分析系统（Mapping Display and Analysis System，MIDAS，现在的 MapInfo）和地球资源数据分析系统（Earth Resource Data Analysis System，ERDAS）。还有两个公共领域系统（MOSS 和 GRASS GIS），它们分别始于 20 世纪 70 年代末和 80 年代初。

有一些空间标准来自 ISO 211 技术委员会（ISO Technical Committee 211，ISO/TC 211）和开放地理空间联盟（Open Geospatial Consortium，OGC）。OGC 是一个国际行业协会，由 384 家公司、政府机构、大学和业界代表组成。他们联合制定了一份公开的地理数据处理规范，基于这份规范可对产品进行符合性测试，测试结果的符合度作为产品分级的依据。主要的目标是信息交换，而不是查询语言，但他们也为不同的查询语言提供 API 接口，并且提供地理标记语言（Geography Markup Language，GML）。http://www.opengeospatial.org/standards/as 是可以查看其当前信息的网页。

在实践中，大多数查询并不是用线性编程来实现的，而是以来自显示屏的图形化方式去完成的。查询结果也是典型的图形学和传统数据混合展示的混合。表 8-1 列出了一些基本的 ISO 标准。

表 8-1　ISO 标准

标准	描述
ISO 19106:2004: Profiles	GML 的简单特性
SO 19107:2003: Spatial Schema	参见 ISO19125 和 ISO19115，在这个标准中定义的基本概念被落实。这是 GML 中简单特性的基础
ISO 19108:2003: Temporal Schema	GIS 中的时态感知（时间件）数据
ISO 19109:2005: Rules for Application Schema	UML 中的概念模式语言
ISO 19136:2007: Geography Markup Language	这相当于 OGC GML3.2.1

8.1　GIS 查询

这么多的历史和工具，GIS 查询与基本的 SQL 查询有哪些不同呢？当我们请求一个典型的 SQL 查询时，我们希望以标量单元的标准对数据进行数字化概要的聚合。而在 GIS 中，我们想要的是一个空间性的答案，而空间既不是线性的也不是标量的。

8.1.1　简单位置

最基础的 GIS 查询是很简单的，如“我现在在哪？”但是你需要的不仅仅是“在这里”

这样的答案。让我们想象一下以下两种定位模型。假设我们在船上或者飞机上，那么答案是与时间和方向有关的，以便预估沿途中各个地点的到达时间。但是，如果我们在灯塔中，意味着我们将会在同一个地方待上非常长的时间，甚至长到经历城市变迁，君士坦丁堡、伊斯坦布尔和拜占庭（其他名字除外）是同一位置不同历史时期的名字。哦，等等！它们的位置随着时间的推移也改变了。

8.1.2 简单距离

一旦你有方法去描述任意的地点，下一步显然就会测算两个地点之间的距离。但是测距并不那么简单。美国人用“乌鸦飞”来描述直连的直线距离，但是距离并不是直线的。

实际距离可以是航线、公路或者铁路的距离。距离可以用旅行的时长来表示。距离也可以是无限。例如，在矿山的相对两侧的两个村庄就是遥不可及的。

8.1.3 在一个区域中查找数量、密度和内容

查询要处理的下一个命题是区域查询。或者说，它的体量是多大？在传统的制图中，用海拔标识第三个维度，并且地形图中都是有等高线的。但是，在 GIS 查询中，第三维度通常是其他元素。

SPIKE 网络上播放的《酒吧营救》（*Bar Rescue*）是以 John Taffer 改造不景气的酒吧为特色的一档电视节目。John Taffer 是这个领域的专家，他通过这个节目给出了很多管理酒吧的学问。几乎每一期节目都有这样的一个环节，Taffer 展开酒吧临近区域的地图并针对客户的情况做 GIS 分析。

第一个查询总是对附近区域的测评。附近有多少其他酒吧（数量）？它们的分布是聚集的还是分散的（密度）？他们中有多少人喜欢你的酒吧（内容）？这些都是静态的测量数据，可以通过人口普查和查询合法数据库的方式获得。Taffer 补充的是经验和分析。

8.1.4 邻近关系

在邻近区域中，邻近关系比测量相邻酒吧的步行距离更加复杂。邻近的概念是基于访问路径的。在纽约，大多数人是步行、乘坐出租车或者乘坐地铁，人行道是访问路径；在洛杉矶，几乎没有人步行，街道是访问路径（如果有停车场）。

在 GIS 中最常见的错误是在地图上画一个圆圈，并假设它是你的邻近区域。大部分 GIS 系统都能够取得这个圆圈内的家庭普查数据。获取这些数据的关键是将圆圈改成多边形。基于你获得的其他信息，扩大或者接触临近区域的边缘。

8.1.5 时间关系

最简单的测量是测线性距离，但是行程时长可能是更重要的参考标准。设想州际高速公

路出口的加油站，它们之间的物理距离可能很长，但是行程时长可以很短。

应急车辆的最短行驶时间的 GIS 查询既不是直线距离，也不是道路的距离。预计出行时间也取决于出行的日子和具体的出行时间。

GIS 的一个最好的应用是图像随着时间动态变化。增长和衰减的模式它都可以展示。通过这些模式，我们可以做预测。

8.2 定位

第一个关于地理学的问题是简单的定位地球上的一个地方并且找出这个地方有什么。现在，我们理所当然地使用全球定位系统（GPS）和精准的地图，但是情况并非总是如此。人类历史的绝大多数时候，人类是依赖于太阳、月亮、星星来获取数据的。地理位置还取决于可能在文本中描述的物理特征。

上面的方法适用于相对较短的距离，但是不适用于较长的航距或者精确度较高的测量。我们需要全球系统。

8.2.1 经度和纬度

经度和纬度对于小学生都是一个很熟悉的概念。设想地球是在不停地绕地轴旋转的球体（事实上不是），在球体上面画假想的线。连接南极和北极的线称为经度，从赤道开始的圆圈称为维度。这个系统可以追溯到公元前 2 世纪，天文学家希帕克斯（Hipparchus）对巴比伦计算进行了改进。巴比伦人使用六十进制的数字，因此我们就有一个角度测量系统，这个系统基于 360° 的圆，分为 60 个单元（参见图 8-1）。希帕克斯使用一年中每天的长度和太阳位置之间的比率，这可以用日晷做到。

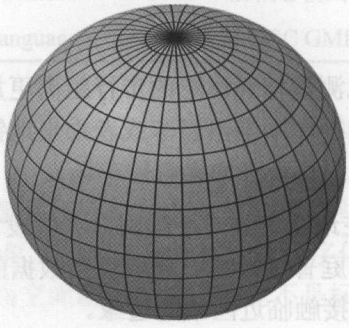


图 8-1　经度和纬度线（来源：http://commons.wikimedia.org/wiki/mage:Sphere_filled_blue.svg。版权所有。GNU Free Documentation license, Creative Commons Attribution ShareAlike license）

经度的确定更困难，最简单且最佳的解决方案是利用精确的时钟。你设置一个格林威治标准时间，获取本地的时区时间，直接计算经度。但是时钟并不准确，在航行过程中并不好使用。在 18 世纪 70 年代，约克郡的一位名叫约翰·哈里森（John Harrison）的木匠发明了航海精密计时器[①]（又称经线仪）。前面的方法都采用月距法，这种方法需要星表（星星什么时间处于什么位置的历表）、太阳、月亮、十颗星星在海上来确定经度，但是它只能精确到半度。由于这些表来源于皇家天文台，格林威治时间正式成为国际标准。

大多数 GIS 问题都足以让我们认为地球是平的。这避免了球面三角学，使我们回到平面几何问题上。我们可以通过调查数据来绘制地图。我们应该更关心代表邻近关系、行政区、道路和其他人造地理特征的多边形和线。

遗憾的是，这些东西在过去并没有很好的测量。在得克萨斯和西部的其他州，界址线在 200 年以前或者更早的时候就使用当时认为有效的设备进行了勘测（大多数就在西班牙/墨西哥时期就完成了）。许多早期的调查使用了经常发生移动的物理特征。河流是最典型的例子，测量标志也已经丢失或者移动。得克萨斯非常大，所以在度角或边界标志上即使是一个小的错误也会导致在边界线之间产生狭窄的楔形无人区。这些楔形的重叠区域在得克萨斯西部地区非常大。它们也成为重叠性区域。

但是，不要以为现代 GPS 就会完全避免这样的错误。GPS 也有错误，只是比传统的测量工具产生的错误少一些而已。在相邻位置上的两个接收者通常会遇到类似的错误，但这些问题常可以被纠正。两个接收者之间的坐标差要比单个接收器的绝对坐标要更准确一些。这就是所谓的*微分定位*（differential positioning）。GPS 会被一些物体封锁，我们说的不是大的物体。网格式栅栏或其他小物体都可以使卫星信号发生偏移。卫星信号通过两个路由才到达接收者时，会发生多重路径错误。同时大气扰动也会对 GPS 产生影响，这种影响类似于对电视卫星天线接收的干扰。

这些调研需要重做。用今天的技术为一小块土地建立边界和多边形边角要远远比 10 年前要容易很多。但是也有法律问题，谁将获得数千英亩地的所有权，修改所有权后的土地将由谁管辖[②]。如果你想获得更多的信息，可通过网站 http://hovell.net/types.htm 查看土地类型调查。

8.2.2 层次三角网格

可用“层次三角网络”（hierarchical triangular mesh，HTM）来替代经纬度。经纬度的基础是在地球表面的二维坐标系统中确立一个点。而 HTM 是通过内置多边形来定位的，地球表

① 参见 https://zh.wikipedia.org/wiki/约翰·哈里森或者 http://baike.baidu.com/view/5454215.htm。——译者注

② 重新测量并修改边界后，必然会导致需要补充土地或是会有土地剩余，那么补充的土地从哪里来，剩余的土地又归属谁？这些都是法律问题。——译者注

面被划分为大小相等的三角形，这些三角形都带有唯一的标识符。HTM 指标优于采用两极奇点坐标的坐标制图方法。

如果看过网格状球顶或者巴克明斯特富勒（Buckminster Fuller）的地图，你就会对 HTM 方法有一些感觉。HTM 从一个零级的八面体开始。要把地球画成八面体（8 个三角形组成的规则多面体），要对它做排列，第一次切分将球体分为南半球和北半球。现在沿着子午线切分，两次切分相互垂直。在每个半球上以“N”或者“S”做前缀，将这些球面三角形从 0 到 3 编号。

平面上的三角形一直是准确的 180°，但是在球体表面和其他凸曲面表面上，它要大于 180°，而在凹曲面上它又小于 180°。如果想要一个快速的思维模型，可以想象一下凸曲面中央有更大的面，一个凹曲面就像马鞍或者喇叭口一样中央部分太小并且是弯曲的形状。

这 8 个球面三角形标记为 N0～N3 和 S0～S3，在系统中称为“零层 trixel”。每个 trixel 能够递归切分成 4 个更小的 trixel。在三角形各边中间放一个点，使用这 3 个点用大圆弧段去构造内嵌三角形。这将会使原三角形变成下一个层的 4 个球面三角形。trixel 切分是递归的，它会递归切分到更小的 trixel，能够切分到任何你想要的层次（如图 8-2 至图 8-4 所示）。

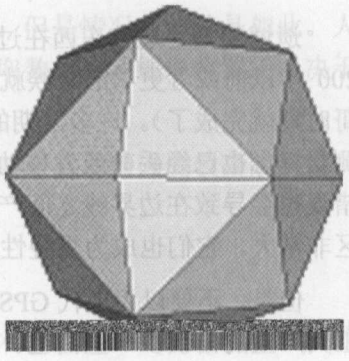


图 8-2　第二层内嵌

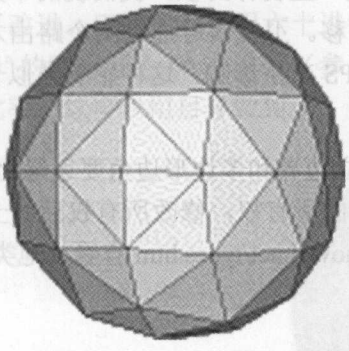


图 8-3　第三层内嵌

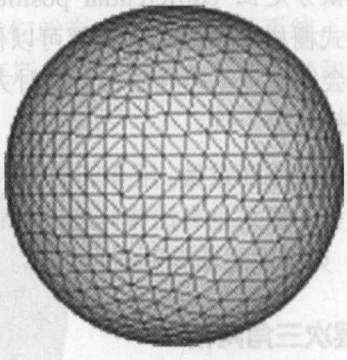


图 8-4　更高层内嵌

要命名新的 trixel，为其使用专有名词 trixel 并且附加数字 0～3，并采用逆时针方式。下一层拐角的点（n={0, 1, 2}）和上一层拐角的编号是相反的。中心的三角形常常命名为 3。

在每一层上三角形的大小几乎是相同的。随着它们变得越来越小，差异也变得越来越小。在第 7 层约有 3/4 的 trixel 比平均尺寸小，有 1/4 的 trixel 比平均尺寸的大。产生这些差别的

原因是 3 个拐角 trixel（0, 1, 2）均小于中心 trixel 3。还记得本节开始时讲的关于空间三角形的几何内容吗？最大与最小区域在更高深度的比值大约是 2。显然，小 trixel 有更长的名字，并且名字的长度也表示了它所在的层次。

在 trixel 中，“名字”可以用来计算精确的位置，位置由给定 trixel 的 3 个顶点向量来表示。可以通过给每一层分配两个位，并将“N”编码为 11，“S”编码为 10，就很简单地将“名字”编码为一个整数标识符。例如，N01 编码为二进制 110001 或者十六进制 0x31。HTM ID（HtmID）就是用唯一的 64 位字符串（但不是所有的字符串都是合法的 trixel）表示的 trixel（及其中心点）的编号。

递归可以是无限的，最小的有效 HtmID 是 8 层，但是可以很轻易地到达 31 层，用 64 位来表示。25 层对于绝大多数应用来说都足够好了。当地球被递归切分到 25 层时，被切分的三角形周长大约是地球表面上的 0.6 米或者是 0.02 弧秒。到 26 层，周长长度大约是地球表面上的 30 厘米（不超过 1 英尺）。如果你想要得到到某一物体更近的距离，你可能是在尝试将一枚子弹射入它或者是要对其进行精细的外科手术。

关键点在于，用 trixel 覆盖一个不规则的多边形。一个 trixel 能够完全包含在一个区域中，也可能完全在一个区域外，或者在区域中重叠，或者与其他区域临接。显然，用越小的 trixel 来近似地表示区域，就越接近这个范围的近似值。一旦假设某个点是 trixel 的中心，就可以采用数学方式或者是构建一个查找表去获得位置之间的距离。深度超过 7，地球的曲率就变得无关紧要了，所以在一个 trixel 中计算距离就是简单的笛卡儿运算。把三角形展平，在起始三角形和终点三角形之间运行一个字符串。你可以在微软研究论文 MSR-TR-2005-123 中获取更多的数学知识。

8.2.3 街道地址

美国和其他一些国家正要将系统迁移至 911 系统。它对街道添加经纬度标识，用于紧急服务。911 系统也由此得名。在这之前，用地址名称表示位置，例如，“约翰逊农场”就表示它在郊区。

数据库称为“主街道地址指南”（Master Street Address Guide，MSAG），它包含关于地址的各方面信息，包括街道名称和门牌号范围的准确拼写。它还关系到美国邮政服务公司（USPS）的编码准确性支持系统（CASS）。可以通过网址 http://pe.usps.com/text/pub28 获取更详尽的 CASS 信息。

美式地址并不通用。2012 年，韩国就爆发了一场大辩论。这场辩论是关于该国地址是否应该从日式转到美式。在日本，乡镇或者城市都会先分成社区。在每个社区里面，街道有自己本地化的名字，并且建筑物会按其被建造的顺序进行编号。就算这栋建筑被拆，它的号码

也不会再用。除非你事先知道哪个地方在哪，否则在这样的系统中几乎不可能使用导航。没有办法从这些地址获取地理信息。

在韩国，反对者主要来自佛教团体。许多社区是用佛教圣人的名字命名的，当这些物理上连接的街道被统一命名时，他们害怕这些名字会消失。

8.2.4　邮政编号

我不能评论地球上所有的邮政地址。所以从文化的基础上，我将目光集中在美国、加拿大和英国。这些都是说英语的国家，有一个共同的传承，所以貌似可以期望有一些共性。但是，事实上并非如此。

影响邮政编码的因素可以大致集中到基于地理和基于邮政投递系统。美国邮政编码系统具有很强的地域，加拿大的系统却较少如此，而英国的系统是基于19世纪的邮局。

8.2.5　ZIP编码

ZIP（zone improvement plan，地区改进计划）编码是一种被USPS使用邮政编码系统。“ZIP编码”是一个由USPS持有的注册服务标志（一种注册商标），但是注册已过期。如今这个词已经成为所有邮政编码的通用名称，几乎没有人记得它原来的意思。基本的格式是一串5位数字，出现在美国州名地址缩写的后面。1983年，在ZIP基础上扩展出的ZIP+4，除了包含ZIP中的5位数字，还包括1个连接号和比ZIP能够表示更精确的地址信息的4个数字。

这种编码是基于地理信息的，这也就意味着同一 Zip 邮编的地理位置是相连的。第一位数字表示连续的几个州。

前3个数字确定邮区（sectional center facility，SCF）。SCF将邮件分类分派给ZIP编码前3位数字相同的所有邮局。邮局大多数时候是按照州区域划分的，但是也有一些特例，例如，会按照军事基地来做划分，或者有些地方是跨跃多个区的行政单位，以及那些可以由离它最近的SCF服务的地区。

最后两个数字表示的是在一个 SCF 内的更小区域。但是这与行政上的区域并不一定吻合，他们往往是为了确定一个当地的邮局或是为一个指定区域服务的邮局驻点。

8.2.6　加拿大邮政编码

加拿大（www.canadapost.ca）是最后一个拥有全国性邮政编码系统的西方国家（1971年）。加拿大的邮政编码是由 6 个字符组成的字符串，是加拿大邮政地址的组成部分。它是一个数字字母编码系统，与美国的 ZIP 编码相比，并没有那么强的地域性。这种邮政编码的格式如下：

```
postal_code SIMILAR TO '[:大写字母:][:数字:][:大写字母:] [:数字:] [:大写字母:][:数字:]'
```

注意，第三个字符和第四个字符之间有空格分隔。前 3 个字符表示一块很大的连续地理区域，称为前向分拣区（forward sortation area，FSA），通常是地理意义上的区分。例如，A0A 在纽芬兰而 Y1A 在育空。

但也并不是总是如此。字母 D、F、I、O、Q 和 U 不允许使用，当快速浏览时，这些字母很容易被误认为数字或者是其他字母。字母 W 和 Z 也不允许作为第一个字母。FSA 的第一个字母表示一个特定的邮政区，除魁北克和安大略以外，一般来说邮政区覆盖范围相当于整个省。但是，由于魁北克和安大略的人口众多，这两个省份分别有 3 个和 5 个邮政区。每一个人口稠密的城市，都有一个专门的邮政区（“H”表示拉瓦尔和蒙特利尔，“M”表示多伦多）。

与此相反的另一个极端是，努纳武特地区和西北地区（Northwest Territories，NWT）非常小，它们共享一个邮政区。FSA 中的数字，用来区分农村和城市，通常 0 表示农村，1 表示城市。FSA 中的第二个字母用来代表一块特定的农村地区、整个中等规模的城市或者大都市地区的一部分。

加拿大邮政编码的后 3 位数字称为本地投递单元（local delivery unit，LDU）。本地投递单元对应一个具体的地址或者一小片地理区域，它可以对应整个小城镇，也可以对应一个中等规模城镇的某个重要组成部分，还可以对应较大的城市街区的一侧，甚至是一个大的建筑物、一个大的机构（如大学或医院）或者收到大量定期邮件的一个企业。以 0 为结尾的 LDU 表示邮政设施，包括邮局和小型邮政超市网点以及分拣中心。

在城市地区，LDU 可能指定的是邮政运营商特定的路线。但是在农村地区，直接送货上门是不现实的。一个 LDU 就代表这一组邮箱或者是一条农村邮政路线。LDU 9Z9 是专门用于商业回复邮件的。在农区地区的 FSA 中，前两个字符的分配是由社区的名字来决定的。

LDU 9Z0 通常指的是大型区域的配送中心设施，也可作为一个占位符，出现在某些区域邮戳中，如在 K0H 9Z0 出现在金斯敦、安大略的本地邮件上。

8.2.7 英国邮政编码

英国邮政编码是由英国国家邮政引入的，从 1959 年到 1974 年服务超过 15 年，编码协议中的一部分遵守英国标准 BS-7666。令人奇怪的是，它们也是普查计数聚合的最低层。英国邮政编码是可变长度的字母数字表，因此很难使它们计算机化。而且这种格式也不适合用来区分分拣中心和地方邮局。在早期的 15 年间，它们似乎是基于不同时代本地名字的缩写，导致皇家邮政编码看起像几十年或者几个世纪之前的。

邮政编码补充了一个新的 Mailsort 系统，这个新的编码系统包含 5 位数字，但只为达到或者超过 4000 封的批量邮件提供服务。批量邮件使用 Mailsort 系统可以获得折扣，但是散装

邮寄则不会。

1．邮政编码格式

英国邮政编码的格式一般由下面的正则表达式给出：

```
(GIR 0AA|[A-PR-UWYZ]([0-9]{1,2}|([A-HK-Y][0-9]|[A-HK-Y][0-9]([0-9]|
[ABEHMNPRV-Y]))| [0-9][A-HJKS-UW]) [0-9][ABD-HJLNP-UW-Z]{2})
```

英国邮政编码用一个空格分成两部分。空格前面的部分是“外部编码”，空格后面的部分是“内部编码”。它是一个层级系统，从左到右看，第一个或者前两个字母代表区域，接下来的数字代表该区域下管理的邮区。每隔邮政编码通常代表街道、街道的一部分或者是一个单独地址。这个特征使英国邮政编码对航路规划软件非常有用。

外部编码用来确定负责邮件的分拣和转运的邮局。外部编码可以进一步分解为区域（是标识124个邮政区域中的一个的字母）和邮区（通常是数字）。内部编码要排除字母I、K、M、O和V，以避免扫描混乱。外部编码类近似于地名缩写（伦敦例外）。例如，“L”是利物浦，“EH”是爱丁堡，“AB”是亚伯丁，“BT”是贝尔法斯特及北爱尔兰。

内部编码的剩余部分是用来分拣邮件到本地邮局的。内部编码可以分成邮政部门（一位数字）和递送点（两个字母）。每个邮编对应的地址可以包含100个商家（每个邮编平均包含15个商家），尽管一个大的商区可能只有一个编码。

2．英国伦敦邮政编码

在伦敦，邮政区的邮政编码是基于1856年的地区邮政系统。它与伦敦自治区的当前边界并不匹配，并且可以跟大伦敦地区一些郡县重叠。邮政编号随意标识在地图上并不是地理因素，而是由历史原因造成的。

大多数伦敦中心地区其实需要更多的类似邮政编码，而不是一种合理有序的模式下的邮政编码。所以他们使用像“EC1A 1AA”这样的邮政编码来弥补不足。一些编码是有政府构建并专门由他们使用的，不会考虑遵循模式。例如，在威斯敏斯特：

- SW1A 0AA，下议院；
- SW1A 0PW，上议院威斯敏斯特宫；
- SW1A 1AA，白金汉宫；
- SW1A 2AA，唐宁街10号，首相和第一财政大臣；
- SW1A 2AB，唐宁街11号，英国财政大臣；
- SW1A 2HQ，英国财政部总部；
- W1a 1AA，广播大厦；

- W1A 1AB，塞尔福里奇百货公司；
- N81 1ER，选举改革协会拥有整个 N81。

也有非地理的邮政编码，例如外部编码 BX，这样如果收件人的地址发生变化，这些邮件也可得到保留。外部编码以 XY 开始表示国内地址有误邮件或者是国际出境邮件。但这并不包括针对 Girobank、北爱尔兰地区、英国军队邮局（BFPO）和海外领土的特殊邮政编码。

总之，这个系统是如此之复杂，以至于你需要软件和皇家邮政局的数据文件（邮政编码地址文件，即 PAF，上面大约有 27 000 000 个商业和住宅的地址）以及专业的软件来处理。皇家邮政局并没有放弃 PAF 的所有权，但是许可商业软件供应商使用，每月更新数据。在英国，大多数地址可以只是一个邮政编码加上门牌号。但是没有一种明显的方式表示地理位置，所以 GIS 必须依赖这些数据文件进行查找，再将查到的数据翻译成地理位置。

8.3 GIS 的 SQL 扩展

1991 年，美国国家科学和技术研究所（National Institute for Science and Technology，NIST）建立 GIS/SQL 工作组，开始进行 GIS 扩展。他们的工作是对地理信息系统在 SQL 数据库上做扩展（http://books.google.com）。

1997 年，开放地理空间联盟（Open Geospatial Consortium，OGC）发表“SQL 的 OpenGIS 简单特性规范”，提出了几种概念性的方法来扩展 SQL，以支持空间数据。这一规范在 ODC 网站上可以查看（http://www.opengis.org/docs/99-049.pdf）。例如，PostGIS 是一个开源软件，可以免费使用，并且基于 OGC 标准的空间数据库扩展了 PostgreSQL 数据库管理系统。SQL Server 2008 也有空间数据支持：有 Oracle Spatial 和 DB2 的 Spatial Extender。新的扩展利用 OGC 标准为数据库添加了空间功能，如距离、面积、交集、并集运算和专业几何数据类型。

坏消息是，他们都有一个面向对象的关系数据库模型附身的感觉。简单的事实是，GIS 不同于关系数据库系统，GIS 的用户界面最好是图形化的，而正如 Codd 博士在他的著名的“12 条规则”里所说，RDBMS 良好的工作依赖于线性编程语言。

总结思考

互联网地图的出现（MapQuest、Google 地图等）和个人 GPS 手机已经使“街道上的人”（双关：普通人）意识到 GIS 可以作为一种日常的数据类型。触摸屏也让在便携式设备上使用 GIS 系统更容易。但当传统的数据放在一个 GIS 数据库中时，会掩盖更复杂用途的可能性。

参考文献

Gray, J. (2004). Szalay, A. S.; Fekete, G.; Mullane, W. O.; Nieto-Santisteban, M. A.; Thakar, A.R., Heber, G.; Rots, A. H.; MSR-TR-2004-32, *There goes the neighborhood: Relational algebra for spatial data search.* http:// research.microsoft.com/pubs/64550/tr-2004-32.pdf.

Open Geospatial Consortium (1997).*Open GIS simple features specifications for SQL.* http://www.opengis. org/ docs/99-049.pdf.

Picquet, C. (1832).*Rapport sur la marche etles effets du cholera dans Pariset Le Département de la Seine*. http://gallica.bnf.fr/ark:/12148/bpt6k842918.

Szalay, A. (2005). Gray, J.; Fekete, G.; Kunszt, P.; Kukol, P. & Thakar A. MSR-TR-2005-123, *Indexing the sphere with the hierarchical triangular mesh*. http://research.microsoft.com/apps/pubs/default.aspx?id=64531.

大数据和云计算

简介

大数据这个术语是由 Forrester Research 在白皮书中提出的，大数据的 4V 指的是：规模（volume）、速度（velocity）、多样性（variety）和可变性（variability）。大数据已经应用在目前我们已经讨论过并且试图协调使用的多种数据库模型的场景下。

在一个叫《呆伯特》的动画片中，有一个尖头发老板宣称“大数据在云上，它来自四面八方，它无所不知”（http://dilbert.com/strips/comic/2012-07-29/）。这个老板对云的理解并不像通常角色设定的那么糟糕。Forrester Research 通过 4 个热词创造了一个概念，并且大受欢迎。请注意，价值（value）、准确（veracity）、确认（validation）和验证（verification）等词并不在这 4V 之列。

第一个 V 是规模（volume），但这不是新特征。多年前我们就已经有 TB 级和 PB 级的 SQL 数据库了，仅看沃尔玛的数据仓库就知道了。2013 年一项来自 IDC 的调查宣称，到 2020 年人们管理的数据规模将是 2009 年的 44 倍多。不过这并不意味着这些数据是集中的。

第二个 V 是数据处理的速度（velocity）。随着通信系统的完善，数据输入输出的速度比以前更快。2013 年，位于得克萨斯州的奥斯汀市被 Google 公司选做部署光纤通信网路的第二个美国城市。我可以得到每日账户交易清单，其中我邮寄的纸质支票只需要一个多小时时间清算，而在 20 世纪 70 年代，美联储还自豪于 24 小时的周转时间。

第三个 V 是数据来源的多样性（variety）。数据来源新增了很多种。现在每个拥有手机、平板电脑或者计算机的人都是一个数据源。在数据库领域中我们遇到的一个问题是，提到分

层数据库系统时，COBOL 程序员总是倾向于将其视为一种单一应用程序，而计算、数据管理和数据呈现都在一个模块里完成。这就是为什么 SQL 语句仍然有将货币符号和用来表示钱的标点符号等临时数据转为要本地格式。现在，无法告知数据在哪台设备上，又是谁会是终端用户。我们比以往更需要具有强一致性特点的松耦合模块。

第四个 V 是数据类型的可变性（variability）。Forrester 认为这个特性指的是数据格式的多变性。我们并不仅是使用简单的结构化数据实现大的规模。从纯字节数的角度来看，视频文件轻易成为我房子里最大的数据源。我们放弃了卫星电视和有线电视，只看网络视频和 DVD。Twitter、电子邮件和其他社交网络工具产生的数据量也同样巨大。到处都是标记语言，而且越来越专用。这就是 ETL 工具销售很火的原因。

想想数据量大和处理速度快的不同。电视营销公司知道在他们播广告时会很快被换台。他们可能低估了数据量和数据速度，但是他们知道这一天正在来临。我们不总是那么奢侈，系统在一个点上的致命错误可能会叠加起来。想象一下你位于俄罗斯的车里雅宾斯克的主发布中心，在 2013 年 2 月 13 日被陨石击中时的情况。你的系统集中度越高，一个突发事件造成的损失就越大。如果这时你只有一个发布中心，那你的公司就无法继续运营了。

另一个问题是大数据的管理。如果传统工具在大数据领域不能继续使用，那么数据模型、数据库管理、数据质量、数据支配和数据库编程等工作如何进行。

大数据的目的，或者至少是卖点，是我们可以通过大量的数据得到对企业很有帮助的有用的观点和信息。不过，就像很多时候敏捷编程变成坏的编程的借口一样，你需要对数据质量有所要求。从统计学的角度来讲，我们说“样本量不能解决样本偏差的问题”，或者说一个随机的数量较小的牛群比一个大的但是病快快牛群好得多。

在许多传统数据库（上文的小牛群）中，人们会观察和清除一些数据，但是许多原始未经加工的大数据是无法直接观察的（上文的大牛群），因为数据量太大。低质量的数据将会过于巨大和集中而导致难以管理。

更糟糕的是，很多数据通常是由机器自动产生的，而不是人参与产生的。我们希望，数据产生时能够一直保持数据的整洁。但是系统学规则中有一条是故障安全系统会在安全处理故障前自身就会发生故障（Gall, 1977）。

相对于“大数据”，公平地讲，你不能假定“传统数据”已经遵循了最佳实践，我会说好的数据实践依然适用，但是它们必须针对大数据模型做出适应和调整。

9.1　对大数据和云计算的疑问

“没有什么比提出一个新的规则更难，因为发明者需要做一些竞争对手在旧条件

下已经做得很好的事情，或者不冷不热的防卫者在新条件下做得同样好的事情。”

——Niccolo Machiavelli

通常情况下，Niccolo 是对的。与其他 IT 新事物一样，关于云计算同样有很多质疑的声音。这些质疑通常都是合理的。我们在旧设备上投入资金后，希望能够从中尽我们所能去榨取一切价值。但更重要的是，我们的想法对旧的术语、旧的概念和已知的过程更适应一些。关于云计算的一些经典质疑在下面几节中会简单介绍。

9.1.1 云计算仅是一种时尚

云计算当然是一种时尚！IT 领域的每个新事物兴起时都是一种风尚的，结构化编程、RDBMS、数据仓库等无不如此。其中的奥妙在于，将好的部分从铺天盖地的宣传中过滤出来。虽然呆伯特的尖头发老板认为：“如果我们把大数据引入服务器中，就能避免破产，所以我们掏钱吧。”而你可能需要更理性一点。

如果是这样的，那么云计算就是一个受欢迎的流行时尚。你的网上银行、亚马逊账户、社交媒体、eBay 或者电子邮箱都已经在你的专属云里。苹果公司和 Google 公司已经开始热情地拥抱云计算，认为这是具有生命力的一项技术革新。云计算是一项正在发展的趋势，而不是过时的趋势。

9.1.2 云计算没有内部数据服务器那么安全

对于某些公司是这样的。在冷战时代我做国防合同时，我们在软件和硬件上都有很多的安全机制。但是，我们不会去派武装部队去看护我们的磁盘驱动器。大部分商店并不是对所有东西都加密。高安全性是一个非常不同的世界。

不过你需要在自己的服务中开发保护工具。不要将未加密的数据放在云上或者内部服务器上。即使在内部服务器上，未加密的数据也可能被破解，阅读关于 T. J. Maxx 公司的信用卡信息泄露丑闻或者其他沸沸扬扬的安全漏洞是非常必要的。很明显，同样不要把加密后的数据和密钥同时放在云上。不要把数据集中放在一个站点，而是分开存储在不同位置的服务器上。当其中一个站点被侵入后，切换到其他站点。

9.1.3 云计算代价高昂

是的，转向云计算的初步成本很高。但是这其中可以做一些权衡。你不需要养一些人专门处理内部服务器。在任何科技领域，人工都是最高的。我们又回到了经典的权衡。但是，如果你正在开始一项新业务，那么购买云空间会比使用自己的硬件便宜很多。

9.1.4 云计算太复杂

那又怎样呢？你最有可能的就是从提供商那里购买云服务。作为服务购买者，你的任务

就是为公司选择正确类型的云计算。这样的目的就是为你的员工和用户在技术层面保持尽可能简单。一个专业的公司会选择雇用专业人员，这与你买了律师服务是一样的道理。

9.1.5　云计算对大公司才有意义

实际上，作为小公司，可以不需要购买昂贵的软件授权和雇用专业人员。有很多例子表明，相当多的小公司通过使用云计算来服务用户。如果成功了，你可以从云服务中转移出来。如果失败了，失败的代价也会很小。

9.1.6　只是技术上的改变

汽车代替马了吗？没有，为汽车设计的城市和为马设计的城市是不一样的，观念陈旧的呆子们需要好好看看 Orsen Welles 的经典电影 *The Magnificent Ambersons*[①]（1942）了。这部电影的故事背景设定在汽车正在改变美国文化的时期，这时汽车不仅仅是技术上的改变，更是文化上的改变。

让我给大家提一个问题：使用云计算后，公司员工如何通过云使用公司内部资源？如果一个在线的服务器发生问题，你可以走到机房去检查硬件。如果云发生错误，你就不能去机房解决问题了，此时你的用户依然抓狂。

当今这个年代已经没有纯技术的改变了，至少公司的律师们总要置身其中。我最喜欢举的一个例子（关于云计算的优缺点）是一个 Kyle Godwin 运行的追踪中西部地区的本地高中和大学体育赛事的网站，该网站将业务数据放在 Megaupload 上。Megaupload 是一个文件共享网站，在 2013 年 1 月被司法部（DOJ）以涉及软件隐私为由关闭。当 Kyle Godwin 试图要回自己的数据时，司法部驳回了他的请求，因为司法部认为这些数据不属于他。截至 2013 年 4 月他的这个案子正在由电子自由基金会（EFF）处理的。

除此之外还有其他几个关于数据归属权的法律问题，所以你需要仔细审视一下你的合同。其中一些如服务器上的电子邮件数据是已经确定归属权的，但是确定之余还是有一些其他未解决的问题。

使用云

让我引用 PabloValerio（2013）中的一篇文章如下。

如果你需要确保自己的数据明确地标识为是你自己的，以避免任何可能的争端，那么接

① 参见 https://zh.wikipedia.org/wiki/安培逊大族。——译者注

下来你需要同云提供商协商一致，你需要足够明智地在合同中提及以下这些条款。

- 明确指出在合同有效期内任何时间点，这些数据归还给你的过程、时限和途径。
- 同时指出归还数据的格式——通常是指数据初次存储时的格式。
- 规定数据全部归还给你的组织时所用的时间限制，通常是天数。
- 明确规定对你对所存储数据的所有权，这样你就不会失去你的数据所有权或者版权。
- 有时我们会只是简单地接受服务提供商提出的条款(在这里我们只能选择同意那些有利于提供商的条款)，而意识不到潜在的问题，所以我郑重建议你咨询律师。

9.1.7 如果网络中断，云计算将毫无用处

这是对的，网络连接是脆弱的。Netfix 就因为他们的服务——AWS 因种种原因导致服务中断而蒙受过损失。

这对大型数据中心的电力供应也是一样的。位于佐治亚州亚特兰大市的美国国家数据中心数十年前就开始负责信用卡信息处理。他们应对一级主电站故障失效的方式是从二级分电站接入电力供应。在经过一场暴风雪后所有二级分电站都失效了，他们又添加了第三级电力供应，并且增加了大型电池储备。

如果整个互联网发生故障，那可能就意味着我们所说的世界末日吧。但是你可以在其他服务商预留一个备用线路。这是一个技术问题，你需要从公司业务停运所造成损失的角度来看待。例如，我的一个视频下载站点丢失了部分配音，并且这是我看的这个系列电视剧的唯一源站，那我可以等一天直到这个故障恢复。但是当一个卖衣服的网站故障不可访问了，我会简单地换个网站消费。

9.2 大数据和数据挖掘

数据挖掘是随着数据仓库（上一个 IT 界热词）兴起的。数据仓库以更利于统计分析和报告的格式集中汇总数据。这导致大量的数据采用星形或者雪花形架构模型——ROLAP、MOLAP 和其他 OLAP 模型的变种。

数据仓库是反范式的，不要指望它能够处理数据，或对数据流有所感知。但是数据仓库中的数据仍然是结构化数据，大数据所指的数据是非结构化的，包含多种数据类型。如果可以从混合的数据中取出结构化数据，那就已经有工具分析它。

9.2.1　用于非传统分析的大数据

政府和企业越来越需要监控你在 Twitter 和 Facebook 发布的复杂类型数据，而不是简单地进行的统计分析。《美国新闻与世界报道》（*U.S. News and World Report*）杂志在 2013 年曾经报道过美国国税局收集纳税人的“巨量”个人信息。这些数据由社保号、信用卡交易和健康记录组成，将在奥巴马医改方案中通过机器进行自动审计。2002 年由 Steven Spielberg 根据 Philip K. Dick 的短篇小说改编的电影《少数派报告》（*Minority Report*）预测过一个新的未来图景，其中一个叫“犯罪预防组织”的警察部门，能够侦查出人类的犯罪企图，进而将其逮捕，虽然逮捕的罪名是人还没有实施的。你只是被假定有罪，并没有进一步的证据。

美国国税局的高级顾问 Dean Silverman 说，国税局将会花时间去审计你的亚马逊账户。这并不是新消息了，RapLeaf 是一家数据挖掘公司，它已经被发现违反隐私条款在社交网络（如 Facebook 和 MySpace）上收集用户个人信息。他们的宣传标语就是美国人 80%的电子邮件都在其网站上有实时数据。这其中的门槛在于处理社交网络的非结构化数据并不容易，你需要像 IBM 的沃森这样的超级计算机去读取并理解这些数据。

2013 年 5 月，美国政府会计办公室发现国税局系统有严重的 IT 安全问题。他们指出，在之前的审计中，美国国税局所做的 118 项系统安全相关的规定只解决了 58 项。并且在进一步的审计中发现，在这 58 个“执行了的”项目中仍有 13 项未完全彻底执行，所以国税局并不是完全符合自己的规定。IRS 的大数据分析不太可能会成功，尤其是当他们不得不开始追踪奥巴马医改的执行情况和惩罚不购买健康保险的人时。

2010 年梅西百货公司仍然使用 Excel 表格分析用户数据。2013 年，macys.com 每天使用上千万 TB 的信息，包括社交媒体、买卖信息，甚至是大数据分析系统提供的信息。他们预计这将是零售业的一个大的提升。

克洛格公司（Kroger）CEO David Dillon 曾经宣称，大数据分析是他与其他零售业对手竞争时的“秘密武器”。零售业依赖于快速周转，薄利多销，以及复杂的库存问题。任何小的提升都是很重要的。

大的零售连锁（希尔斯、梅西百货、沃尔玛等）希望能够实时地应对市场变化，有以下几个目的。

- 根据货物的受欢迎程度来动态标价和动态配货。这里一个明显的例子是季节性商品，如圣诞树在 7 月肯定卖的不好。但是其他商品就需要精细调整，如 7 月的时候应该在美国的哪个部分以何种价格卖何种类型的泳衣。
- 在收银台向用户推销。这意味着用户数据分析必须在用户刷卡前完成。
- 更严格的库存控制以避免货物积压。这个目标的实现需要引入外部数据，如天气预报

或者社交媒体以及已经收集到的内部数据一起分析。天气预报告诉我们什么时间往芝加哥发送多少雨伞。社交媒体可以告诉我们应该往芝加哥发送什么类型的雨伞。

零售商的敌人是在线零售网站：鼠标点击对实体销售。在线零售网站通过不同方法使用大数据，亚马逊网站发明了现代用户推荐模型。起初这个模型是很粗糙的，需要手动调整。那时我最喜欢的个人经验就是确信如果有用户买了一本晦涩难懂而少人问津的数学书的话，那么该用户应该也会喜欢一个奇怪牌子的休闲裤（但是我穿套装）。相信我，我是那个月地球上唯一一个买那本书的人。今天我的 Netflix 和亚马逊的个人推荐基本上是我曾经读过、看过或买过的了，这意味着我的个人信息是准确匹配的，因此当我看到一个我之前没见过的推荐时，我会对这个推荐很有信心。

9.2.2 系统合并的大数据

阿肯色州公共服务部（DHS）在一个架构比较老的系统中有 30 个分离的数据系统，缺乏一个用户视角的能够纵观全局的视图。他们试图安装一个新系统，能将州内的社会项目，包括医疗补助、补充营养协助计划（Supplemental Nutrition Assistance Program，SNAP）以及州儿童健康保险计划（State Children's Health Insurance Program，SCHIP）等，融合在其中。这种合并将会是跨部门协作的，所以会同时出现政策问题和技术问题。

这个目标是建立一个能够让用户访问所有适用计划的入口，以及一个当有变化发生时不管用户享用多少福利计划都能够通知所有有关机构的信息源。

出于同样的目的，伊利诺伊州公共服务部（DHS）决定将所有文件数字化并通过大数据模型管理。2010 年 DHS 在地方机构和数据中心拥有超过 1 亿页纸的文件。无法立刻将所有文件都数字化并存入云中，代价高昂且工作量巨大。相反，该局决定刚开始通过 3 张基本的用于处理申请的表开始，并以 PDF 文件格式按时间顺序存储记录。该州正在使用 IBM 的企业内容管理大数据技术。当用户联系该机构时，工作人员进行一系列的询问并将答案记录到在线表格中。基于这种信息，这个系统确定计划是否合格，分配元数据，并以电子表格方式存在中心仓库中。工作人员花费在搜索信息上的时间从数日降到了几秒，这对于用户服务来说是一个突破。DHS 的 CIO Doug Kasamis 说该系统在 3 个月内创造的价值就抵消了投入。

总结思考

一项大数据云服务提供商 Infochimps 在 2013 年做的调查显示，81%的受访者将大数据/高级分析项目视为 2013 年 IT 领域的 Top 5 发展项目。然而，受访者同样宣称 55%的大数据项目并没有完工，许多其他项目因为反对声而功亏一篑。我们首先根据 Gartner 关于“炒作周期”的理论，大数据将在 2013 年 1 月达到其“膨胀预期的峰值”。这种情况准确发生在上一

个 IT 领域热词“数据仓库”上。大数据的失败也同样是由于过于膨胀。纯粹数据是不能成功的，这是一个管理失误。

但是，人们仍旧信任数据仓库，因为它不会将数据暴露给外界。在 2013 年年中，我们开始追查奥巴马政府通过“棱镜门”计划到底在美国人身上进行了多大范围的监控。这些监控都是通过云来监视邮件、社交媒体、Twitter 和几乎其他所有媒介。这一结果已经使大数据在隐私方面的信誉受损。

参考文献

Adams, S. (2012). *Dilbert*. http://dilbert.com/strips/comic/2012-07-29/.

Gall, J. (1977). *Systemantics: How systems work and especially how they fail*. Orlando, FL: Quadrangle. ISBN: 978-0812906745.

Gartner Research. (2010–2013). http://www.gartner.com/technology/ research/hype-cycles/.

McClatchy-Tribune Information Services (2013). *Illinois DHS digitizes forms, leverages mainframe technology*. http://cloud-computing.tmcnet.com/news/2013/03/18/6999006.htm.

Satran, R. (2013). *IRS High-Tech tools track your digital footprints*. http://money.usnews.com/money/personal-finance/mutual-funds/articles/2013/04/04/irs-high-tech-tools-track-your-digital-footprints.

Valerio, P. (2012). *How to decide what (& what not) to put in the cloud*. http://www.techpageone.com/ technology/how-to-decide-what-what-not-to-put-in-the-cloud/#. Uhe1PdJOOSo.

Valerio, P. (2013). *Staking ownership of cloud data*. http://www.techpageone.com/technology/staking-ownership-of-cloud-data/#.Uhe1QtJOOSo.

生物特征、指纹和专业数据库

简介

当前，生物特征还没有商业化应用。它们通过*生物实体*而非商业实体来识别人。我们处在一个医学和法制的世界。所以，当安全性成为一个话题时，生物特征才可能会转到商业领域，我们希望的是安全地交易这些私密信息。

汽车有交通工具识别码（Vehicle Identification Numbers，VIN），图书有国际标准图书码（International Standard Book Number，ISBN），公司有邓氏编码（Data Universal Numbering System，DUNS），零售商品有通用商品条形码（Universal Product Code，UPC）。但是，人并不是组装起来的货物，并没有 ANSI/ISO 标准。

SQL 数据库论坛上一个标准钓鱼攻击问题就是如何用自然特征来识别人。反钓鱼攻击会拒绝任何与个人在特定数据模型中扮演的角色相关联的标识符，如账号。它是神奇的、通用的“人标识符”，它 100%正确、经济实用且立即可用的。当然，需要指出的是，现在并没有一个为所有人准备的神奇的、通用的数据库，所以“人标识符”不现实且可能会违反隐私法律，自然也就无法阻止钓鱼攻击了。

人类是生物学意义上的商品！人类有基于生物特征的生物特征标识符。遇到的第一个问题是收集这些原始数据，即生物特征测量值。很显然，解决这个问题需要人体和一些仪器。第二个问题是如何编码这些测量值，以便我们能够将其保存在数据库中并进行查找。

人是多维度的复杂生物。因为没有一个共同的标准，所以到现在为止还没有一种明显的方式来对人进行分类。在你想要帮自己的同伴辨识出人群中的某个人时，你可能会用这样的

语言描述："他/她看起来<像少数民族/很温柔/有一定岁数了>"。如果这个人从统计学意义上来说不寻常并且群体规模比较小，这种方法是行得通的。Wilt Chamberlain 将其自传命名为 *Just Like Any Other 7-Foot Black Millionaire Who Lives Next Door*，仿佛他已然泯然于某个群体中。但是从另一个极端来说，对于模特而言，年龄和种族的模糊往往是一个优势。

10.1　原生生物特征

第一个生物特征标识符是面部识别。人类能够通过观察识别其他人，但是将其计算机化却很困难。人类凭借的是大脑中模糊的逻辑，并且对于识别人没有准确的度量单位，这种"看脸识人"的感觉已经被根植到人类的大脑中。如果大脑执行这个重要工作的部分受损了，你就会变成"脸盲"，无法将你邻座的人与他的照片对上号，甚至无法认出镜子中自己的影像。

在计算机时代之前，最开始的一些识别人类的方法是打标签、纹身和为罪犯或一些组织的成员进行身体标记。想想这些类似于条形码的解决方案实在是太差了，它们使用的设备非常糟糕。接下来，人类通过视觉记忆和相册来识别人。但是，人会随着时间改变，体重会变化，胡须会随着时间和流行方向改变，并且岁月也会在人身上留下痕迹。流行杂志喜欢将名人的学校纪念册上的旧照片与当前照片进行比较。但是，我们仍然会接受将糟糕的驾照相片作为一个有效的身份证明。

数据库需要的是一个编码系统，而不是人的描述。大约在 1870 年，法国人类学家 Alphonse Bertillon 设计了一个由 3 部分组成的识别系统。第一部分是身体的某些部位的骨骼尺寸的记录。这些测量值被缩减为一个公式，理论上，只适用于一个人，并且这些测量值在他或她的成年生活中不会改变。其他两部分是正式的描述和我们今天仍然使用的"面部照片"（见图 10-1）。

这个系统还引入了在卡片上保存数据的概念，称作 Bertillon 卡，它可以根据特征排序，并且能代替纸质档案被快速检索。一个训练有素、经验丰富的用户能将成百上千的卡片减少到一个小平台就能放下的候选人，这样一个人就能对犯罪嫌疑人或照片进行对比。Bertillon 卡利用边缘上的小孔来让视觉上的排序更容易些。

Bertillon 卡编码了囚犯的眼睛、耳朵、嘴唇、胡子、头发颜色、肤色、种族、额头、鼻子、身材、下巴、头型、体毛、眉毛、眼球及眼眶、嘴、外貌、脖子、肩部倾斜度、态度、举止、嗓音及语言还有衣着。

Bertillon 系统被广泛接受已经超过 30 年。由于测量他或她的特征需要一段时间，所以通常它被用来决定拘留的嫌疑犯是惯犯还是重复的嫌疑人。这对于我们在电视节目上观看的犯罪科学调查（crime scene investigations，CSI）并没有什么用。

图 10-1 Bertillon 的识别系统

Bertillon 系统的衍生是面部识别系统、手掌几何特征识别和其他生物特征识别系统的基础。现代系统是基于可从静态图片或者视频构建的特征比率的，而不是试着将人简化为数字。

10.2 指纹

指纹被用于识别要回溯到巴比伦时代和泥板文书。它们在古代中国、日本和印度被作为

合同签名使用。但是直到 1788 年，德国解剖学家 Johann Christoph Andreas Mayer 才发现对于每个人来说指纹是独一无二的。

收集指纹比制作 Bertillon 卡容易多了。即使在今天，通过计算机化的存储，执法机关依然是使用十指卡片来保存十个手指的信息。正如数据库人员所预料的，问题在于缺乏一个分类系统。根据生活的地域的不同这里有几个不同的选项。最流行的十指扫描分类系统包括 Roscher 系统、Juan Vucetich 系统和 Henry 系统。Roscher 系统是在德国研发的，并在日本和德国实施。Vucetich 系统是在阿根廷研发的，现在仍然在南美使用。Henry 系统是在印度研发的，并在大多数英语国家使用，包括美国。如今，它通常被称为 Galton-Henry 分类，这是因为这是由 Francis Galton 爵士根据 Henny 系统完成的。

10.2.1　分类

在最初的 Galton 分类系统中，有 3 个基本的指纹样式：箕型（60%～65%）、斗型（30%～35%）和弓型（5%）。根据这个基本模型，按照指纹尾部指向手的方向，我们还能分出一些子类，弓型分为弧形的弓型或者帐篷形的弓型，箕型分为尺骨形的箕型和放射形的箕型。尺骨形箕型开始于小指的位置，更接近尺骨一些，比臂骨低一些。反箕型开始于拇指的位置，更接近桡骨一些。斗型也有一些子分类组，包括弧形的斗型、杂形的斗型、双箕形的斗型、孔雀眼斗型、双斗型、囊形的斗型。除此之外还有弧形的弓型、帐篷形的弓型和囊形的箕型。

现代系统应该会让数据库使用人员感到高兴，因为它只使用了一个简单的散列算法。它包括 5 个部分，*R* 代表右，*L* 代表左，*i* 代表食指，*m* 代表中指，*t* 代表大拇指，*r* 表示无名指，*p* 代表小指。编码方式为 *Ri*/*Rt*+*Rr*/*Rm*+*Lt*/*Rp*+*Lm*/*Li*+*Lp*/*Lr*。每个指纹被赋的数值是根据它们是否是斗型来决定的。第一个部分的斗型值为 16，第二个部分为 8，第三个部分为 4，第四个部分为 2，最后一个部分为 0。弓型和箕型被赋为 0。按照下面的方式将分子和分母相加：

$$(Ri+Rr+Lt+Lm+Lp)/(Rt+Rm+Rp+Li+Lr)$$

分子分母同时加 1，是为了避免被 0 除的情况。例如，如果右无名指和左食指为斗型，将编码如下：

$$0/0+8/0+0/0+0/2+0/0+1/1$$

计算结果为：

$$(0+8+0+0+0+1)/(0+0+0+2+0+1)=9/3=3$$

这样就只有散列值为 3 的指纹才能匹配这个人。

10.2.2 匹配

匹配一个指纹与将其分类不一样。通过在十指卡滚动每根手指来得到手指的每一面。在真实世界里，与数据库比对的样本一般是不完整的、弄脏的或者损坏的。

这意味着作为第一个过滤器搜索必须以分类开始，然后由技术人员对纹路进行计数，最后的比对手动完成。曾经，IBM 制造过一个特别的设备，面板上有 10 个旋转式的转盘开关，每一个对应着每个手指的纹路。这对于不懂计算机的警方人员比较容易操作。

指纹图像系统如今使用了不同的技术——光学、超声波、电容或者热效应——来测量纹路的不同。机器可以被分为两个主要的分支：固态指纹读取器和光学指纹读取器。真正的问题是，人是柔软的，所以每个图像会因压力、皮肤状况、温度和其他采样噪声而失真。

为了克服这些问题，我们现在使用非接触式的三维指纹扫描仪。图像都是数字化的。我们现在非常擅长高分辨率图像处理，能够通过比对纹路之间的距离得到一个比例来进行调整，这样就能将失真的图像恢复为原样。

自从指纹被用于安全性和访问筛选，这些系统通常使用以前存储的模板和一个候选指纹。算法找到图像中的一个或多个点并与候选指纹进行匹配。这个信息对硬件来说是本地的，并不涉及数据交换。大数据数据库花时间进行匹配，所以我们试着优化算法和数据库硬件。当前的硬件每秒大约能匹配 1000 个指纹。

2013 年 4 月，Safe Gun Technology（SGTi）公司表示有望在接下来的两个月内开始生产自己的智能枪。Columbus，一家基于遗传算法的公司，通过定位在武器控制器上的一个平面的、红外的读取器来使用相对简单的指纹识别方式。生物特征读取器支持其他 3 种物理机制，分别是扳机、撞针、击锤。控制器芯片可以保存 15 000～20 000 个指纹。如果一个大型军事组织想将成千上万的指纹定制到单个武器中，这也是可行的。单支枪的所有者也可以临时将朋友或者家人的指纹定制到枪里来射击目标，回到家后再将其删除即可。

10.2.3 NIST 标准

NIST（National Institute for Science and Technology，美国国家科学和技术研究所）作为 ANSI 和 ISO 的成员设定标准已经数十年了。它们处理的不仅仅是指纹：它们有设定很多指纹、掌纹、脚底、面部、虹膜和其他身体部分，并且还有伤痕、记号和纹身（SMT）的标准。这个标准所使用的记号，意思是针刺记号或者药物使用的痕迹。

这个标准的第一版是 ANSI/NBS-ICST 1-1986，是一个指纹标准。随着时间它被逐渐完善，在 1993 年、1997 年、2000 年和 2007 年均被重新修订。2008 年，该标准采纳了使用可扩展标记语言（XML）的遵循 NIEM 的编码方式。NIEM（National Information Exchange Model，

国家信息交换模型）是美国司法部和国土安全部之间的一种合作关系。对于相关的用户而言，这份文档最有用的部分是标识符类型及其记录的列表，如表 10-1 所示。

表 10-1　标识符类型及其记录

记录标识符	记录内容
1	交易信息
2	自定义描述性文字
3	低解析度灰度指纹图像（已废弃）
4	高解析度灰度指纹图像
5	低解析度二进制指纹图像（已废弃）
6	高解析度二进制指纹图像（已废弃）
7	自定义图像
8	签名图像
9	特征数据
10	面部、其他身体部分或 SMT 图像
11	语音数据（未来增加）
12	牙科记录数据（未来增加）
13	可变解析度的指印摩擦嵴图像
14	可变解析度的指纹图像
15	可变解析度的掌纹图像
16	自定义可变解析度测试图像
17	虹膜图像
18	DNA 数据
19	可变解析脚掌图像
20	信源表示
21	与上下文相关的
22～97	预留给将来使用
98	信息安全
99	通用生物识别交换格式框架（CBEFF）生物识别数据记录

请注意语音数据和牙医记录并不是这些标准的一部分。它们经常出现在电视上的罪案节目中，但是实际上，它们非常罕见，并没有被添加的价值。在大多数情况下，牙医记录被用

来在死后识别一具尸体。语音很难进行匹配，目前还没有一个语音数据库可以检索。

类型4的记录是以标称500ppi（像素/英寸）扫描出来的单一指纹图像。需要14条类型4的记录在一个文件中组成经典的十指卡片（全部10个卷起的单个指头，2个拇指的印记，以及每只手的4个指头的2个同时印记）。我们可以将一些类型14的记录组成指纹图像。

类型18的记录是关于DNA的。它使用了ISO/IEC 19794-14:2013信息技术——生物特征数据交换格式——第14部分：DNA数据标准。出于隐私方面的考虑，这个标准只使用非编码区域的DNA，并且避免了其他区域的显性信息。更多DNA的内容将在10.3节详细介绍。

正如我们所期望的，我们有很多3字母的缩略短语：SAP（对象获取概要）是一组生物识别特征。这些概要都有一些帮助记忆的方法：脸部的SAP、指纹（fingerprint）的FAP和虹膜（iris）记录的IAP。面部图像的类型10记录强制包含SAP代码，FAP在类型14记录中是可选的，IAP在类型17记录中是可选的。

转移到机器处理是很重要的。人类太容易出错，并且对于大型数据库来说处理速度太慢了。1995年，协同测试服务（Collaborative Testing Service，CTS）组织了一次水平测试，在第一次时，由国际鉴定协会（International Association for Identification，IAI）“设计、组织和检查”来看看训练有素的人是如何处理实际数据的。

结果令人失望。4张嫌疑人卡片，上面有与其他7份混淆指纹混在一起的10个手指的指纹。一共156人参加了测试，只有68人（44%）正确地区分了所有7个混淆的指纹。总之，测试包含了一共48个错误的识别结果。*Journal of Forensic Identificaiton* 的编辑David Grieve描述了法医对CTS的测试结果的反应为“从震惊变为难以置信”，并说：

> 该学科如此之高的错误率极大地反讽了当初所宣扬的绝对准确的识别，而这样的结果是令人心灰意冷的。有34个参与者完全用错误结果替代了真实情况，让人难以置信的是他们占总体参与者的22%。不管采用哪种方式，这样的结果都代表了无法让人接受的现实写照，提高正确率需要整个学术团体的积极努力。

10.3 DNA识别

DNA（脱氧核糖核酸）图谱并不是全基因组测序。DNA图谱足以完成亲子鉴定和犯罪证据的测试。值得庆幸的是，DNA测试能够被编码为一系列数字，而这可以作为人的标识符。尽管对于每个人来说，99.9%的人类DNA序列都是一样的，并且人和黑猩猩差异度不到2%，但这还是足以将一个人和另一个人区分开的。

在同卵（“完全一样”）双胞胎的案例中，在基因层面还是会有足够的差异性（http://www.scientificamerican.com/article.cfm?id=identical-twins-genes-are-notidentical）。在基因分化点，一

对双胞胎将会基于相同的基因而产生一些不同的副本，这种遗传状态称为副本数变异。通常，人的每个基因会携带两个副本，它们来自于双亲。但是，还是会有一些基因组区域能携带来自于任何地方的 0～14 份基因副本。

我们已经在电视或报纸中见过和读过亲子鉴定测试，所以对亲子鉴定测试都很熟悉了。最初的测试能快速地根据母亲和孩子的样本排除父亲。表 10-2 是一份商业 DNA 亲子鉴定测试报告样例，它使用了 5 个 DNA 标记。

表 10-2　亲子鉴定报告样例

DNA 标记	母亲	孩子	指称的父亲
D21S11	28, 30	28, 31	29, 31
D7S820	9, 10	10, 11	11, 12
TH01	14, 15	14, 16	15, 16
D13S317	7, 8	7, 9	8, 9
D19S433	14, 16.2	14, 15	15, 17

报告显示，指称的父亲的 DNA 这 5 个标记都匹配，那么这位父亲应该是真的。完整的测试结果需要展示孩子与指称的父亲之间在 16 个 DNA 标记上的匹配情况，才能得出这个人是否是孩子生物学意义上的父亲的结论。最初的测试可能会识别出亲近的男性亲属。对于人类而言，完整的基因组在 23 对染色体之上包含大概 20 000 个基因。绘制它们是耗时耗力的。

基本原则和技术

DNA 图谱使用了重复的序列，它们高度变异，称为*可变数目串联重复序列*（variable number tandem repeat，VNTR），特别的，短的重复序列称为 STR。VNTR 位点在近亲之间非常相似，但是由于高度变异，所以没有亲属关系的个体拥有相同的 VNTR 是极不可能的。

这种 DNA 图谱分析技术是 1984 年由 Alec Jeffreys 爵士首次在英格兰的莱斯特大学提到的，目前是一些国家 DNA 数据库的基础。帝国化学工业公司（ImperialChemical Industries，ICI）在英格兰建立了一个验血中心，随之 1987 年 Jeffreys 博士的基因图谱分析技术开始商用。

这项技术的目标是个性化医疗，而非识别。识别并不需要完整的基因组，所以它成本更低。为了想出降低成本的方法，第一个完整的人类基因组测序由人类基因组项目耗费 30 亿美元，最终在 2003 年完成。2010 年起，我们花费不到 1000 美元就能识别特定疾病的标记和基因特质。狗只需 100 美元就能完成识别，这已经被挑剔的[①]附近的屋主协会用来标记狗和它们

① 原文用的是 anal，这个词其实有“肛门的”“挑剔枝节的”等意思，这里作者应该是一语双关。——译者注

的粪便，以此来规范那些不清理自己宠物粪便的主人。

尽管现在每个国家使用基于 STR 的 DNA 图谱系统，但是它们使用的并不都是相同的系统。在北美，CODIS 13 核心位点几乎是通用的，但是英国使用的是 SGM+11 位点系统（它可以与它们国家的 DNA 数据库兼容）。在使用的 STR 区域集合中有一些重叠，这是因为几个 STR 区域可以同时进行测试。

如今，我们可以从几家公司里获取微芯片（松下、IBM、富士等），它可以在一小时之内完成 DNA 图谱分析。芯片的大小在一个硬币之内并且只需要一滴血或其他体液就可以工作。从血液中通过亚微观的“纳米孔”将 DNA 从芯片中抽取出来。芯片中会进行一系列的聚合酶链反应，它可以通过接口被读取。

10.4 面部数据库

我们看着电视上的犯罪秀节目长大，视频监控摄像机录下了坏人，警察拿着视频中的一帧回到实验室用它与坏人的面部数据库进行比对。脸部照片在巨大的屏幕飞速闪过，其速度是如此之快以至于你根本不能认出任何人，直到弹出一个完美匹配的结果，然后情节得以继续。视频图像是如此美丽而光辉，好人也是如此美丽而光辉。案子总是会被解决，除非我们想要看到扣人心弦的下一季。

面部数据库不是那么工作的。就像其他生物特征数据，它被用在有本地数据库的安全系统里，或者是作为中央数据库的一部分被使用。虽然是几乎相同的技术，但是数据库人员并不特别关心个人安全性的使用。

识别算法分为两个主要的方法：几何学的和光度的。光度算法基本是尽量覆盖图片以进行匹配。当你拥有一个大型数据库可以检索时，很容易得到多个匹配结果。这种方法其实是将人用眼睛完成的工作自动化。人脑有一个部分专门负责识别人脸，所以我们为这个生存特质进化出一个复杂的流程（“嗨，你和我不是一伙的”）。但是，有些人患有一种脑部疾病，称为“脸盲”症，他们就不能识别人脸。好吧，我跑题了。

几何学算法从对象脸部的图像中提取标识或者特征。这些特征，如眼睛的中心、鼻尖、颧骨等，都可以被规范化后再被压缩为一个有数学意义的值。“测试图像”与那些压缩的脸部数据进行比较。这有点像散列算法或 CRC 代码——你得到一些过滤后的候选匹配结果。

虽然不存在单一的算法，如果想了解更多的技术细节，可以通过组合使用特征脸、线性判别分析、使用费舍尔人脸算法的弹性束图匹配方法、隐马尔可夫链模型、使用张量表示的多重线性子空间学习和神经元动态激励连接匹配等进行分析。

坏消息是人脸是三维的，不像指纹是平面的。在传统的方法中，观察角度、灯光、口罩、

帽子和发型造成了“信噪”，你在电视节目中看到的那些完美的匹配并不是每次都会发生的。它们是基于静态的、铺平的、8×10 的演员大头照进行处理的！

10.4.1　历史

该领域最早的工作是由 Woody Bledsoe、Helen Chan Wolf、Charles Bisson 在 20 世纪 60 年代中期为一个情报机构完成的。操作员通过使用早期的图形输入板在面部照片上标记坐标和 20 的间距。通过训练，他们能做到为每户采集大约 40 张照片。识别算法其实是通过使用将嫌疑人的一系列间距和数据库记录进行简单比对，返回最相近的匹配结果。

真正的问题在于标准化数据，将脸放置为一个标准的方向。程序需要决定倾斜度、弯曲度、旋转角度，然后使用投影几何进行调整。引用 Bledsoe（1996）的话：“实际上，同一个人头部的两个不同的角度的两张图片的相似度很低。”标准化假定点和距离能够被分配给一个“标准的头部”。这个标准头部是从 7 个真实头部的测量结果里得出的。

如今，产品使用 80 个节点和更专业的算法，由软件测量的节点如下所示：

- 两眼之间的距离；
- 鼻子的宽度；
- 眼窝深度；
- 颧骨的形状；
- 下颌长度。

现在还没有明确的标准，但是 Identix，一个商业公司，推出了一个名为 FaceIt®的产品。这个产品能够从三维图像中生成一个“脸印”。FaceIt 目前使用 3 个不同的模板来确认或识别对象：向量、局部特征分析和表面纹理分析。通过从三维图像中选择 3 个特定的点，就可以同二维图像进行比较。脸印可以以数字化的方式被存储到计算机中。他们现在通过使用皮肤纹理的独特性来获取更多精确结果。

这个过程，称为表面纹理分析（surface texture analysis，STA），工作原理大致与面部识别相同。对某块皮肤拍摄的照片称为肤印。这块肤印会被分为更小块。肤印不仅有肤色，还有纹路、毛孔和肤质。通过它可以识别同卵双胞胎之间的区别，如果只使用面部识别软件是不可能做到这样的。

向量模板非常小巧，主要是用来快速搜索整个数据库从而满足一对多搜索的需求。你可以认为它是一种高级的针对面部建立的索引。而局部特征分析（local feature analysis，LFA）模板是对向量模板的有序匹配结果进行二次搜索。你可以认为它是在总过滤完成后的二级索引。

STA 是 3 种方法中最复杂的。它在 LFA 模板搜索完成后还会根据图中的皮肤纹理执行一次最后的查询，因为图像中包含了最详细的信息。

通过结合这 3 个模板，FaceIt 相对来说对表情的改变不是特别敏感，包括眨眼、皱眉或者微笑，并且还能抵消胡须生长和戴眼镜带来的影响。这个系统还能无差别识别不同种族和性别。

如今，传感器可以捕捉到面部形状及其特征。对于独立的个体来说，眼眶的轮廓、鼻子和下巴是唯一的。想想怪物史莱克，他是真皮肤包裹的三维三角形网格框架。这个框架能被旋转和弯曲，即使这样，也仍然能辨认出这是史莱克。史莱克非常著名，但是这需要复杂的传感器为数据库进行面部捕捉和探测图像。皮肤纹理分析是较新的工具，它根据皮肤的可视化细节作为所获取的标准数据或者扫描图像。这在识别中能够带来 20%～25%的性能提升。

10.4.2 谁在使用面部数据库

首先想到的是，这样的数据库只会被赌场用于来寻找骗子和罪犯，或者用于政府警察机构。这种应用会有，但是也有一些普通的商业应用。

Google 公司的 Picasa 数字图像软件在 3.5 以后的版本有一个内置的面部识别系统。它能将人脸与人联系起来，所以对图片的查询能够返回特定人群的所有图片。

索尼公司的画面移动浏览器（Picture Motion Browser，PMB）可以分析照片，并将照片和相同的脸联系起来，这样他们可以对应地被标记，找出照片与某个人、某些人或无名小卒的区别。

Windows Live Photo Gallery 也包含面部识别。

识别系统也在赌场被用来抓获老千和其他黑名单上的人。

警察的应用程序并不局限于仅仅在调查中使用面部数据库。

- 伦敦纽汉自治区试图在其自治区范围的闭路电视系统里尝试面部识别系统。
- 德国联邦刑事警察局从 2006 年开始已经为全德国警察局配备可以脸部图片进行面部识别的系统。欧盟也有类似的系统，在自愿的基础上，应用在德国和奥地利的国际机场和其他边境通过点的自动边境控制。他们的系统比较个人的面部与电子护照微芯片里的图像。
- 美国州和联邦的执法机构使用被捕人员的面部照片数据库。美国国务院使用的是全世界最大的面部识别系统之一，拥有 7 500 万张照片，被广泛用于签证流程。
- US-VISIT（美国访客暨移民身份显示系统）是针对获得准入美国资格的外国游客。当

外国旅客在收到自己的护照时，需要提交指纹并拍摄照片。指纹和照片将会和一个保存已知罪犯和疑似恐怖分子信息的数据库进行比对。当游客到达美国某个港口的入口时，这些相同的指纹和照片会用来验证这个想要通过入口的人和收到签证的人是否为同一个人。

- 墨西哥在2000年总统选举使用面部识别来防止选举舞弊。这是种防止两次投票的方法。
- 在2001年1月的超级碗，当时警察在佛罗里达州的坦帕湾使用了Viisage面部识别软件来搜索观众中潜在的罪犯和恐怖分子。潜在确定了19个有少量案底的人。

ATM机和个人计算机的面部识别已经经过测试，但是还没有广泛应用。安卓手机有个被称作Visidon Applock的应用。该应用允许手机用户对自己的应用添加面部识别锁。面部识别技术对于Macintosh的iPhoto应用来说已经实现了。另一个目的是为了那些患脸盲症的人，这样他们就能认出他们的同伴了。智能相机能够侦测到的不仅是对焦，还有闭着的眼睛、红色的眼睛和其他影响照片效果的情况。

10.4.3　它有多好

坦白地说，这并不是最明显的生物特征。DNA和指纹更加可靠、高效并且易于检索。面部识别主要的优点是它不需要对象的许可或物理样本就能在人群中发现他们。Ralph Gross——卡耐基-梅隆大学机器人研究所的研究员，描述了障碍物和观察面部的角度之间的关系："面部识别技术对于正面整个脸部以及有20度角的时候有很好的效果，但是一旦你用于轮廓识别，还是存在一些问题"。实际上，如果你有嫌疑人清晰的正面图像，那么就有92%的可能性能在超过100万个面部图像中找到他/她。

尽管有几个系统的数据库包含的罪犯居住在该自治区里，但是这么多年来前面提到的伦敦纽汉自治区CCTV系统从来没有成功辨认出某个罪犯。但是无处不在的摄像头还是减少了犯罪。同样的效应也发生在佛罗里达的坦帕市，波士顿罗根机场的某个系统在一个为期两年的测试期里不能进行任何匹配后就被关掉了。

在2012年首映的电视节目《疑犯追凶》基于这样一个前提：我们的英雄有一个超级AI程序，能黑进每台计算机、每个监视器，并且通过这个超级AI，能算出某个人将会陷入困境，这样英雄能在节目结束前拯救他们。在2012年，FBI发起了一个价值10亿美元的面部识别程序项目，称为下一代识别（Next-Generation Identificaiton，NGI）项目，并在几个州进行了试点。

FBI的目标是构建一个包括1200万条来自联邦罪案记录和美国国务院的护照签证数据库中的面部照片、声纹和虹膜扫描信息的数据库。它们还能增加来自30多个州的DMV信息，这些州目前保存了这些数据。

我们还没有达到“电视幻想级别”的技术，并且很多年内都可能不会实现。但是，一个由 NIST 提供的称为面部识别大挑战（FRGC）（http://www.nist.gov/itl/iad/ig/frgc.cfm）测试套件，从 2004 年 5 月一直运行到 2006 年 3 月。更新的算法比 2002 年的面部识别算法准确 10 倍，比 1995 年的准确 100 倍。其中的一些算法甚至能够识别出人类都识别不出的同卵双胞胎。

尽管官方的数据库里有 Dzhokhar Tsarnaev 和 Tamerlan Tsarnaev 的图像，但是 FBI 的面部识别系统还是没能阻止 2013 年的马拉松爆炸案。FBI 保存的 Tsarnaevs 兄弟的图像是摄像头从远处拍到拍摄的，照片很模糊，他们还戴着帽子和太阳镜。调查人员观看监控录像，看见他们正在放置炸弹，从而发现了他们，接着通过剑桥市的一个 7-11 便利店里面的安保摄像头拍摄的图像匹配到了面部。在这些照片被公布于众之后，Facebook 和数以百计的手机摄像头填补了信息的空白。

总结思考

生物特征将会在不久的将来克服技术问题。可以匹配 DNA 和指纹的廉价设备甚至是智能卡的出现将成为可能。识别即将到来。真正的问题是政治而不是技术。

参考文献

Grieve, D. (2005). *Fingerprints: Scientific proof or just a matter of opinion?* www.sundayherald.com, http://www.zoominfo.com/CachedPage/?archive_id=0&page_id=1324386827&page_url=//www.sundayherald.com/52920&page_last_updated=2005-11-21T04:47:08&firstName=David&lastName=Grieve.

Gross, R. (2001); Shi, J.; Cohn, J. F. *Quo vadis face recognition?* http://www.ri.cmu.edu/pub_files/pub3/ gross_ralph_2001_1/gross_ralph_2001_1.pdf.

Bledsoe, W. W. (1968).*Semiautomatic facial recognition*. Technical Report SRI Project 6693,Menlo Park, CA: Stanford Research Institute.

Bledsoe, W. W. (1966). *Man-machine facial recognition: Report on a large-scale experiment*. Technical Report PRI 22, Palo Alto, CA: Panoramic Research, Inc.

Bledsoe, W. W. (1964). *The model method in facial recognition*. Technical Report PRI 15, PaloAlto, CA: Panoramic Research, Inc.

分析型数据库

简介

传统的 SQL 数据库是用于联机事务处理（OLTP）。它的目标是为日常业务应用提供支持。它们的硬件非常昂贵以至于不能为特殊用途（如分析数据等）专门准备一台机器。但此一时，彼一时。如今，我们有联机分析处理（OLAP）数据库，它是基于 OLTP 的数据的。

这些产品使用数据库在某一时间点的快照，然后将这些数据放入一个多维模型。这个模型的目的是运行处理聚合的数据而不是进行独立事务的查询。它是分析型，而不是事务型。

在传统的文件系统和数据库中，使用索引、散列和其他手段进行数据访问。现在我们仍然使用那些工具，但是我们增加了一个新工具。星形模型、雪花模型和多维的存储方法都是更快获取数据的方法，但是它们是通过聚合而非逐行进行处理的。

11.1 数据立方体

这种结构是数据立方体（超立方体）。想想一个二维的交叉表或者电子表格的那种模型，如位置（编码为东、西、南、北）和产品（用类别进行编码）。网格可以显示位置和产品所有可能的组合，但是许多单元格是空的——你不会在南方销售皮毛大衣，或者在北方销售比基尼。

现在扩展到更多的维度，如支付方式、购买时间等，网格就会变成一个数据立方体，然后是超立方体，等等。如果你不能对超过三维的数据立方体进行可视化，就想象你在环绕立体系统看到的控制面板。每个滑动开关控制声音的某一个方面，如平衡、音量、低音、高音。这些维度是相互独立的，但是这些维度定义了整体。

就像你能看到的，实际上显示一个完整的数据立方体代价非常高昂并且大多数单元格是空的。幸好，我们有从科学编程和很多数据库访问方式总结的使用稀疏数组的经验。

OLAP 数据立方体是从星形模型生成的，这个我们接下来会讨论到。中间的是事实表，它列出了组成查询的核心事实。一般来说，星形模型的事实表通过与维表相关联模仿了稀疏数组的单元格。星形模型是非标准化的，但是由于数据从不更新，所以不会得到异常并且无需加锁。事实表的行存储了一个完整的事实，例如一次购买（谁、什么、何时、怎么做等）。而维表提供了度量的单位（例如，购买行为可以按照周、年、月、购物季等进行聚合）。

11.2　Codd 博士的 OLAP 规则

E. F. Codd 博士和他的合作者在 1993 年出版了 Hyperion 公司解决方案相关的白皮书（见 Arbor 软件），名为 Providing OLAP to User-Analysts: An IT Mandate。这里面介绍了 OLAP 术语和一系列抽象的规则，有点像关于 RDBMS 的规则。但是，由于论文已经被一个商业供应商赞助（它的产品和这些规则契合得非常好），所以与像他的 RDBMS 这样的纯粹的研究工作相比，它并没有被很好地接受。

Codd 博士被指控在论文中署名，但他并没有做太多工作，而是由供应商、他的妻子和一位研究助手完成。最初的论文有 12 条规则，在 1995 年又增加了 6 条规则。这些规则被重构为 4 个特征集合，在这里会进行总结。

为 Codd 博士说句公道话，他用易于理解的抽象概念首次定义一个新技术的方法并引导每一个追随他的数据库革新者发表他们的第一篇公开论文。只是到了后来，你才能看见只有专家才看得懂的学术数学论文。他对于 OLAP 的观点也随着时间推移而建立起来。

11.2.1　Codd 博士理论的基础特性

Codd 博士理论的原始编号被保留下来了，但是包含了更多的内容。

- F1：多维概念视图。这意味着数据被保存在一个矩阵中，矩阵的每一个维度都是一个属性。这是一个网格和电子表格数据模型。它有在维度上基于限制条件对数据进行选择的能力。
- F2：直观的数据操作。直观是一个含糊的术语，每个供应商都这样声称他们的产品。这通常意味着你有一个可进行拖拽的图形用户界面（GUI）和其他图形化界面。并没有排除某种书面编程语言，但是它并不会给你带来设计方面的帮助。
- F3：可访问性。OLAP 作为介质。OLAP 引擎是在可能的异构数据源和 OLAP 前端之间的中间件。你可能想将这个与 SQL 模型进行比较，SQL 引擎是在用户和数据库

之间的。

- F4：分批提取与解释执行。OLAP 必须有为 OLAP 数据准备的自己的中间数据库，还要可以实时访问外部数据。我们在不久在后面会讨论到 HOLAP（混合 OLAP）。实时访问外部数据是一个重要的问题，这是因为它意味着各种连接和可能的大量意想不到的数据流入中间数据库。
- F5：OLAP 分析模型。Codd 博士在他的白皮书中描述了 4 个分析模型。
  - 分类的：从数据处理开始，这就是一个报告中典型的描述性统计。
  - 解释的：这就是我们对电子表格所做的——切片、切块和下钻来按需生成报表。
  - 深思的：这是“假设”分析。有一些专门的工具用来针对这种分析模型进行建模，并且它们中的一些工具还扩展了电子表格。深思分析让你对整个系统变化的影响有了疑问，例如“关闭 Alaska 的商店对公司有什么影响？”换句话说，你想对你的某个特殊想法进行测试。
  - 公式化的：这些是目标搜索模型。你知道你想要的结果，但是你不知道如何才能得到它们。这种模型持续不断地改变参数并且进行思考才能得到想要的结果（或者证明目标是不可能达到的）。你需要设定一个目标，如“我怎样才能提高 Alaska 商店的比基尼销量？”然后等待答案。坏消息是可能有很多种解决方案、没有解决方案（“比基尼在 Alaska 是注定要失败的”），或不可接受的解决方案（“关掉除了 Alaska 的所有商店”）。

分类和解释特性很容易实现。深思和公式化特性实现起来比较困难并且代价高昂。我们有一些针对工业过程使用线性或约束编程的公式化分析和深思分析的经验。但是参数的个数必须要少，很好控制，并且结果要很好量化。

这个缺点导致了“模糊”的逻辑和数学模型，它们的数据不需要传统意义上的精确并且能基于大数据集进行快速响应（豹纹在 Alaska 通常卖得好，所以这个夏天或许可以在 Alaska 销售豹纹比基尼）。

- F6：客户端服务器架构。这是不言而喻的。目标就是使用户可以轻易分享数据并且能够使用前端工具。如今，这都是基于云的访问。
- F7：透明。这是 RDBMS 的一部分。客户端前端不需要知道如何连接 OLAP 引擎或者其他数据源是如何生成的。
- F8：多用户支持。这是也是 RDBMS 模型的一部分。这实际上很容易，因为 OLAP 引擎是只读的数据快照。不需要对多用户提供事务控制。但是，有一种新的分析型数据

库，其设计理念是可以实时查询从外部数据源流入的数据。

11.2.2　独有特性

这个额外的特性清单使 OLAP 引擎变得实用。

- F9：*对非标准化数据的处理*。这意味着我们可以从非 RDBMS 数据源加载数据。SQL-99 标准的第 9 部分，也增加了为外部数据准备的 SQL/MED（外部数据的管理）特性（CAN/CSA-ISO/IEC 9075-9-02，2003 年 3 月 6 日，ISO/IEC 9075-9:2001 采用，第 1 版，2001 年 5 月 15 日）。这个提议并不遥远，但是当前的 ETL 产品可以用它们特有的语法来处理这些转换过程。
- F10：*存储* OLAP *结果*。这实际上是一个实用的考量。获取 OLAP 数据的代价是高昂的，你并不希望从实时数据中不停地重构。另外，这意味着 OLAP 数据库是数据源状态的快照。
- F11：*提取缺失值*。在他的第二版关系型模型（RMV2），Codd 定义了两种缺失值，这不是 SQL 中使用的 NULL。其中一个类似于 NULL，这模型是属性存在于实体中，但我们并不知道它的值。第二种缺失值表示属性并没有存在于实体中，所以它根本不会有值。由于大多数 SQL 产品（First SQL 例外）都只支持第一种缺失值 NULL，所以它难以满足规则 F11。但是，还是有些支持 CUBE 和 ROLLUP 特性来决定哪些 NULL 值是存在于原始数据，哪些 NULL 值是在聚合中生成。
- F12：*处理缺失值*。所有的缺失值。所有的缺失值不论什么来源都被 OLAP 分析器忽略了。这遵循了标准 SQL 中在进行聚合时丢弃 NULL 值的规则。但是在统计学中，还是有一些方法是围绕缺失值进行处理的。例如，如果已知的值服从正态分布，系统就能估计出在这个正态分布中的缺失值的值。

11.2.3　报表特性

报表特性显然是整个 OLAP 的重点，不可缺失。但是这感觉比理论更加商业化。

- F13：*弹性报表*。这个特性又有点模糊。我们可以将其理解为维度可以被聚合与排列从而得到用户想看到的几乎任何数据。如今，这通常意味着提供漂亮的图形的能力。可视化现在变成了 IT 的一个独立领域。
- F14：*一致的报表性能*。Codd 博士要求报表性能不能随着维度数量和数据库规模的增长而显著下降。与抽象原则相比，这更像是一个产品设计目标。如果你有预先计算好的数据库，那么维度的数量就不是大问题。
- F15：*物理级别的自适应*。Codd 博士要求 OLAP 系统要自动适应它的物理存储。这可

以和大多数产品中实用程序一起完成，这样用户可以控制自适应选项。

11.2.4 维度控制

这些特性可以处理数据立方体的维度，并且我们可以将其和以上特性一起使用。

- F16：通用维度。Codd 博士采用纯粹的观点，每个维度必须与它的结构和运算能力等价。这或许不是不与事实表相连，这是一个多维数据库的特性。但是，他确实把额外运算能力分配给了被选择的维度（假定包括时间），但是他坚持这些额外的功能应该对任意维度都是可授予的。他并不想要基础数据结构、公式或者报表格式偏向任意一个维度。

  这条后来被证实为是最初的 12 条规则中最有争议性的一条（当这些特性被重新修订后，它被重新编号了）。技术型产品很大程度上倾向于遵守它，所以这些产品的供应商支持它。应用型产品通常并不遵守，所以它们的供应商猛烈地抨击这条规则。用一个严格而纯粹的解释来说，几乎没有产品完全遵守它。我建议，如果你想购买一个工具来应对通用的目的或者供多个应用使用，再来考虑这条规则。但是即使这样，它的优先级也比较低。如果你想要买一个产品应对特定的应用，你完全可以忽略这条规则。

- F17：没有限制的维度和聚合等级。这在物理上是不可能的，所以我们可以勉强认为是“大量”的维度和聚合等级。Codd 博士建议产品应该支持至少 15 个维度，20 个更好。经验法则告诉我们准备超过现在需要的空间以应对将来的扩展。

- F18：无限制的交叉维度操作。计算和操作之间还是有区别的。特定规模的组合不能用于同样的计算从而得到一个有意义的结果（例如，“周四除以红色是什么？”是无意义的）。但是，对混合的数据进行一个操作是可能的（例如，“周四我们卖了多少双鞋？”）。

11.3 MOLAP

MOLAP，或者多维 OLAP，是 Codd 博士在他论文中描述的“网格中的数据”版本。这是存储汇总结果的观点的第一次出现。一般来说，MOLAP 在更小型的数据库上执行更简单的计算的速度比较快。业务用户在过去几十年有着良好的电子表格设计技术，同样的技术可以被用于 MOLAP 引擎。

电子表格模型是大多数人学的第一种（通常也是唯一一种）声明式语言。它的优势在于世界上电子表格的使用者远远多于 SQL 程序员。他们无需为它学习一种新的概念框架，而只需要学习一种新的语言即可。

11.4　ROLAP

ROLAP，或者说是关系型 OLAP，是 MOLAP 之后发展起来的一种 OLAP。其与 MOLAPDE 的主要区别在于 ROLAP 无需预先在数据库进行计算或者存储汇总数据。当用户请求数据时，ROLAP 工具生成动态 SQL 查询。要说有什么例外的话就是物化视图的使用，它会按照第一次调用时的查询持久化汇总数据。ROLAP 的目标之一是可扩展性，这是因为它会减少存储需求，还有就是要更加灵活并且可移植，这是因为它使用了 SQL 或者类 SQL 语言。

另外一个优势在于 OLAP 的引擎可以和事务型数据库的引擎相同。RDBMS 引擎在处理大量数据、并行工作、优化查询存储在它们内部的特定格式方面非常在行。丢掉这些优势是非常可惜的。例如，DB2 的优化器现在可以通过查询一个连接很多小表的单独大表从而侦测到星形模式。如果它发现了一个星形模型，它就会基于这个假设生成合适的执行计划。

11.5　HOLAP

纯粹的 ROLAP 引擎的问题在于它要慢于 MOLAP。想想大多数人实际上是如何工作的。宽泛的查询被缩窄的更加具体。特定的表，如一个普通的汇总表，可以一次生成并在多个用户之间共享。

这就导致了 HOLAP 和混合 OLAP，它将结果表保存在专门的存储设备中或对这些表建立索引，这样它们就能够被复用。这些基础表、维度表和一些汇总表均保存在 RDBMS 里。这在当今的产品里是最常见的方法了。

11.6　OLAP 查询语言

对于事务型数据库来说，SQL 是标准查询语言。除了一些被添加至 SQL-99 标准的 OLAP 特性以外，没有一种语言用于分析。最接近的是微软的 MDX 语言，由于微软市场的领导地位，它已经成为了事实上的标准。

这并不是因为 MDX 是一种卓越的技术语言，而是因为微软使其比其他产品更为便宜。它的语法混合了 SQL 和某种面向对象的方言。与那些完整的统计学软件包，它还是不够强大。

如 SAS 和 IBM 的 SPSS 等的统计学语言，已经存在了数十年。这些产品有很多选项和强大的计算能力。事实上，你真的需要成为专家才能完全掌握它们。如今，它们有了图像化界面，这使得编码更加容易。但是这并不意味着你不需要通过统计学知识来做出正确的决策。

11.7 SQL 中的聚合操作符

当 OLAP 进入 IT 领域时，SQL 委员会想要取领先的地位。不过有个大问题，就是供应商会基于专有的语法和语义创建独有的特性。这会导致各种方言并存并且基于标准的已经完成的工作被混淆。所以 SQL 需要 OLAP 函数。

OLAP 函数新增的 ROLLUP 和 CUBE 部分是 GROUP BY 子句的扩展。ROLLUP 和 CUBE 扩展通常称为超集。可以通过使用 GROUP BY 和 UNION 操作符的旧的标准 SQL 编写它们，但是这样一来，它会变得错综复杂。为旧操作定义一个新特性作为简化是很有必要的。这样编译器的作者能重用他们已有的部分代码并且程序员可以重用他们已有的部分思维模型。

但是对于 SQL，这也意味着这些新特性的结果也将是 SQL 表，而不是一种新的数据结构，如经典的 GROUP BY 结果集。

11.7.1 GROUP BY GROUPING SET

从 SQL-99 开始，GROUPING SET (<column list>)是报表中的一系列通用的 UNION 查询的简化。例如，求总量：

```
SELECT dept_name, CAST(NULL AS CHAR(10)) AS job_title, COUNT(*)
 FROM Personnel
 GROUP BY dept_name
UNION ALL
SELECT CAST(NULL AS CHAR(8)) AS dept_name, job_title, COUNT(*)
 FROM Personnel
 GROUP BY job_title;
```

可以重写为：

```
SELECT dept_name, job_title, COUNT(*)
 FROM Personnel
 GROUP BY GROUPING SET (dept_name, job_title);
```

对于所有的分组函数来说存在一个问题。它们会为求和级别上的每个维度生成 NULL 值。如何才能区分原始数据中真正的 NULL 值和生成的 NULL 值？这就是 GROUPING()函数的工作。原始数据的 NULL 值返回为 0，生成的 NULL 值返回为 1，这表示分类汇总。

这里有个小窍门可以得到人类可读的输出：

```
SELECT CASE GROUPING(dept_name)
 WHEN 1 THEN 'department total'
 ELSE dept_name END AS dept_name,
 CASE GROUPING(job_title)
```

```
  WHEN 1 THEN 'job total'
  ELSE job_title_name END AS job_title
  FROM Personnel
GROUP BY GROUPING SETS (dept_name, job_title);
```

实际上这是一个糟糕的编程实例，这是因为显示应该在前端而不是在数据库中完成。另一个问题是你可能想要在查询中使用 ORDER BY，而不是得到一个随机顺序的报表。但是我们在 SQL 中不关心这个问题。

组集的概念也能用来定义其他 OLAP 组。

11.7.2 ROLLUP

ROLLUP 组是对 GROUP BY 子句的扩展，它生成了一个包括除常规分组行的分类汇总行的结果集。分类汇总行也就是超聚合行，它包含了通过应用相同列函数得到的进一步的聚合值，这些列函数是用来获取分好组的行。ROLLUP 分组是一系列组集。

下面是一个用经典 COBOL 语言写的“控制中断报表”，并且报表开发人员使用的是顺序文件处理：

```
GROUP BY ROLLUP (a, b, c)
```

等价于

```
GROUP BY GROUPING SETS
(a, b, c)
(a, b)
(a)
()
```

注意，ROLLUP 的 n 个元素转换为 n+1 组集。还有一点需要牢记，分组表达式中的顺序是特定的，并且还是 ROLLUP 的象征。

ROLLUP 是基于经典的汇总和分类汇总报表，最终以 SQL 表的形式呈现。下面的例子是 3 个销售区域的简单报表。ROLLUP 函数被用于 GROUP BY 子句中：

```
SELECT B.region_nbr, S.city_id, SUM(S.sale_amt) AS total_sales
 FROM SalesFacts AS S, MarketLookup AS M
 WHERe S.city_id = B.city_id
 AND B.region_nbr IN (1, 2, 6)
GROUP BY ROLLUP(B.region_nbr, S.city_id)
ORDER BY B.region_nbr, S.city_id;
```

SELECT 语句用法与平常一样。FROM 语句生成了一个执行表，WHERE 子句移出了不满足搜索条件的行，GROUP BY 语句将数据分组后将其化简为一行聚合值并且将列进行分组。

表 11-1 所示为 SQL 的结果样例。结果显示，两个分组（区域、城市）的 ROLLUP 返回 3 个合计，包括区域、城市和总计。

表 11-1 城市和区域的年度销售额

区域号	城市 ID	销售总额及说明
1	1	81
2	2	13
…	…	…
2	NULL	1123 -region #2
3	11	63
3	12	110
…	…	…
3	NULL	1212 -region #3
6	35	55
6	74	13
…	…	…
6	NULL	902 -region #6
NULL	NULL	3237 -grand total

11.7.3 CUBE

CUBE 超集是 SQL-99 对 GROUP BY 的另一个扩展，它产生了一个包含 ROLLUP 聚合的所有分类汇总行，以及交叉分组行的结果集合。交叉分组行是额外的超聚合行。顾名思义，当数据以电子表格的形式呈现时，按照交叉列进行汇总。就像 ROLLUP 一样，CUBE 组也可以被认为是一系列组集。对 CUBE 来说，CUBE 分组表达式的所有组合结果将会单独计算总量。所以，CUBE 的 n 个元素将转化为 $2n$ 个组集。例如：

```
GROUP BY CUBE (a, b, c)
```

等价于

```
GROUP BY GROUPING SETS
(a, b, c) (a, b) (a, c) (b, c) (a) (b) (c) ()
```

注意，CUBE 的 3 个元素转化为 8 个组集。与 ROLLUP 不同，指定元素的顺序对 CUBE 没有影响：CUBE(a, b)和 CUBE(b, a)相同（见表 11-2）。但是可能不会以相同的顺序生成行，这一切取决于你的产品。

表 11-2　性别和种族

性别代码	种族代码	总数和说明
M	Asian	14
M	White	12
M	Black	10
F	Asian	16
F	White	11
F	Black	10
M	NULL	36 – Male Tot
F	NULL	37 – Female Tot
NULL	Asian	30 –Asian Tot
NULL	White	23–White Tot
NULL	Black	20 –Black Tot
NULL	NULL	73– Grand Tot

COBE 是对 ROLLUP 函数的扩展。CUBE 函数不但提供 ROLLUP 中的列汇总，还可以计算行汇总和不同维度的总计。这是统计学中包括的交叉表的一种版本。例如：

```
SELECT sex_code, race_code, COUNT(*) AS total
 FROM Census
 GROUP BY CUBE(sex_code, race:code);
```

11.7.4　用法须知

如果你的 SQL 支持这些特性，需要测试下来，看看 GROUPING() 如何作用于那些由外连接生成的 NULL 值。记住，SQL 不一定以特定顺序返回表中的行。你仍然需要将这些结果放到 CURSOR 中，通过 ORDER BY 子句来生成报表。但是，你可能发现返回的结果是有序的，这是因为 SQL 引擎做了这部分工作。

在这之前，早期版本的 SQL 通过一种隐蔽的方式来进行 GROUP BY 操作。后来，SQL 产品使用了并行处理、散列，以及其他方法来分组，分好的组没有排序等附加效果。我们要总是写标准 SQL 并且不依赖于某一个特定产品的某一个特定版本的内在特性。

11.8　SQL 中的 OLAP 操作符

IBM 和 Oracle 在 1999 年年初共同建议扩展，由于 ANSI 罕见地迅速采取了行动，它们成

了 SQL-99 标准的一部分。IBM 在 DB2 UDB 6.2 中实现了部分规范，在 1999 年中期 DB2 UDB 6.2 以某些形式开始商用。在 1999 年末发布的 Oracle 8i 第二个版本和 DB2 UDB 7.1 包含了上述的大部分特性。

其他供应商也纷纷参与，其中包括数据库工具供应商 Brio、MicroStrategy、Cognos 和数据库供应商 Informix（当时它还不属于 IBM），以及其他供应商。Hamid Pirahesh 博士领导的隶属于 IBM Almaden 研究实验室的一个团队在其中扮演了一个尤其重要的角色。他的团队耗时一年研究了这个课题并想出了一个如何在这个领域扩展 SQL 的方法，然后，他通知了 Oracle。Oracle 了解这个课题后独立完成了其中的一些重要工作。Andy Witkowski 在 Oracle 中扮演了一个关键的角色，这两个公司花费了大约 2 个月的时间，最终敲定了一个联合标准提议。在这个标准提出之前，Red Brick 实际上是第一个实现这个功能的产品，但是并不完整。读者可以通过 ANSI 文档“介绍 OLAP 函数”了解详情，该文档由 Fred Zemke、Krishna Kulkarni、Andy Witkowski 和 Bob Lyle 共同完成。

11.8.1 OLAP 功能

OLAP 函数和 GROUP BY 家族略有不同。你可以指定一个“窗口”来定义被聚合函数作用的行和作用的顺序。当使用列函数时，相对于当前行，适用行可以被进一步优化，正如当前行之前或者之后的一系列行。例如，通过按月的分区，均值可以通过前 3 个月被计算出来。

1. 行编号

当 SQL 基于无序集合时，人们依靠排序来获取数据。你想有一个随机的电话本吗？在 SQL-92 中，为结果集增加行编号的唯一办法是使用游标（实际上是将结果集放置在顺序文件中）或专有特性。供应商的特性是千差万别的。

第一种方法是使用表附加的一个虚列，该列为每一行添加一个自动增加的整数。SQL Server 中的 IDENTITY 列是最常见的例子。第一个实际的考虑是自增长是特有的并且不可移植，所以你要知道当你更换版本或将系统移植到其他产品上时需要修复这些问题。但实际上初学者认为他们从不移植任何代码！也许他们在为快要倒闭的公司工作并且很快就要离职。也许他们的代码是如此之差，没有人想要他们的应用。

但是，让我们来看看这个逻辑问题。首先，创建一个两列表，然后使其自增长。如果你不能通过定义将多个列声明为确定的数据类型，那么自增长根本就不是数据类型。这是物理表的属性而非表中的逻辑数据。

接下来，创建一个单列自增长的表。现在试着插入、更新和删除不同的数字。如果不能进行表中行的插入、更新和删除，那么它就不是一个符合定义的表。

最后，当你用一个 SELECT 语句进行插入操作时，其顺序已经无法预测，代码如下：

```
INSERT INTO Foobar (a, b, c)
SELECT x, y, z
 FROM Floob;
```

由于查询的结果是一个表，并且这个表是一个无序的集合，那么自增长编号应该是什么？集合会完整地、全部地呈现在 Foobar 表中，而不是一次一行。对 n 行进行编号的方式共有 n!种，那么选择哪一种？答案是：不管怎样选择产生结果集的那个物理顺序。这又是非关系短语的“物理顺序”！

但是实际上情况要比这糟糕。如果执行同样的查询，但是如果查询采用了新的统计量或在索引被新增、删除后，新的执行计划可能会使结果集以不同的物理顺序返回。索引和统计并不是逻辑模型的一部分。

第二种方法是以某种编码格式展示磁盘上的物理位置，以便能够用来直接将读/写头移动到记录中。这是 Oracle 的 ROWID。如果磁盘进行碎片整理，那么位置可能会发生变化，可是不能移植代码。这个方法需要依赖硬件。

第三种方法是一个函数。这个函数最初是在 Sybase SQL Anywhere（见 WATCOM SQL）中实现的，并且它是标准 SQL 中的 ROW_NUMBER() 函数的模型。

这个函数在排序语句（如果指定某一列）定义的窗口内计算行的顺序行号，以第一行为 1 开始并且循序增加直到窗口内的最后一行。如果某个排序语句如 ORDER BY，没有在窗口中被指定，行号将被以任意顺序赋给子查询返回的行。在实际的代码中，编号函数被用来展示目的而非在前端增加行号。

一个求中位数的有趣的技巧是使用两个带有 OVER() 语句的 ROW_NUMBER() 函数：

```
SELECT AVG(x),
 ROW_NUMBER() OVER(ORDER BY x ASC) AS hi,
 ROW_NUMBER() OVER(ORDER BY x DESC) AS lo
 FROM Foobar
 WHERE hi IN (lo, lo+1, lo-1);
```

这既处理奇数也处理偶数。如果行数为奇数，那么 hi=lo。如果行数为偶数，那么我们需要中间挨着的两行的两个值。处理 x 列的两个值，并得到一个加权中位数，这样能更好地体现中间的趋势。

```
x hi lo
==========
1 1 7
1 2 6
2 3 5
3 4 4 <= median - 4.0
```

```
3 5 3
3 6 2
3 7 1
```

获取行数为偶数的中位数的例子如下：

```
x hi lo
===========
1 1 6
1 2 5
2 3 4 <= median
3 4 3 <= median=3.5
3 5 2
3 6 1
```

2. RANK 和 DENSE_RANK

到目前为止，我们讨论了扩展一般 SQL 的聚合函数。有些特殊函数可以被用来生成窗口。

RANK 赋予了窗口中行的顺序排名。某一行的 RANK 被定义为 1 加上该行之前的行数。没有区别的行在窗口中的顺序则被赋予相同的排名。如果排序出现两行以上的相同情况，那么在顺序的排名编号中会有一个或多个间隔。也就是说，由于多个相同值，RANK 的结果集在编号中可能会出现间隔。例如：

```
x RANK
========
1 1
2 2
2 2
3 4
3 4
3 4
3 4
3 4
3 4
```

DENSE_RANK 也是对窗口中的行赋予一个顺序的排名。但是，某一行的 DENSE_RANK 是 1 加上在这个顺序中该行之前不同的行的数量。所以，顺序排名编号中，不会有间隔，而且一样的数据被赋予相同的排名。RANK 和 DENSE_RANK 需要一个 ORDER BY 子句。例如：

```
x DENSE_RANK
=============
1 1
2 2
2 2
3 3
```

```
3 3
3 3
3 3
3 3
```

除了这些函数，定义一个窗口的能力对于SQL的OLAP功能来说同样重要。你使用窗口来定义行的集合，行是函数应用的地方和排序发生的地方。另一个了解窗口的概念是将其等同于切片的概念。换句话说，窗口是整体数据域的简单切片。

此外，在使用一个对列作用的OLAP列函数时，如AVG()、SUM()、MIN()或MAX()，目标行可以基于当前行进一步优化，相对于当前行之前或之后的一系列行。关键是用户可以要求将整个SQL语法和以OLAP为中心的SQL语句结合起来。

3．窗口子句

窗口子句有3个子单元：分区、排序和聚合分组。其大致的格式是：

```
<aggregate function> OVER ([PARTITION BY<column list>] ORDER BY<sort
column list> [<aggregation grouping>])
```

一系列列名指定了分区，它应用于先前的FROM、WHERE、GROUP BY和HAVING子句产生的行。如果没有指定分区，那么所有行的整个集合组成一个单一分区并且每次都将聚合函数应用于整个集合。尽管分区看起来像GROUP BY，但是它们并不一样。GROUP BY是将一个分区的行拆分为单一的行，而窗口内的分区只是简单地组织行到组里，并没有拆分它们。

窗口子句内的排序与游标中的ORDER BY子句类似。它包含了一些排序键和是否升序或降序的指示。重要的是理解排序是发生在每个分区内的。

<aggregation grouping>定义了在分区内作用于每一行的聚合函数之上的一系列行。因此，在例子中，对每个月来说，你指定的集合包括这一行和前两行。下面是一个基于SQL-99特性的ANSI论文的例子：

```
SELECT SH.region, SH.month, SH.sales,
 AVG(SH.sales)
 OVER (PARTITION BY SH.region
 ORDER BY SH.month ASC
 ROWS 2 PRECEDING)
 AS moving_average
 FROM SalesHistory AS SH;
```

这里，AVG(SH.sales) OVER (PARTITION BY...)是OLAP函数。OVER()子句里面的结构定义了数据的“窗口”，在此应用聚合函数（本例中是AVG()）。

窗口子句定义了一系列被分好区的应用聚合函数的行。窗口子句表示，采用SalesHistory

表并对其进行以下操作：

- 按照区域字段进行分区；
- 在每个分区内按照月进行排序；
- 将同一区域中的每一行和它的前两行进行分组；
- 在每一组中计算移动平均值。

数据库引擎不需要按照这里描述的顺序来执行这些步骤。但是如果执行了这些步骤，那么必须产生相同的结果。

聚合分组有两种主要类型：物理的和逻辑的。在物理分组里，你对当前行之前和之后指定数量的行进行计数。SalesHistory 表的例子就是使用了物理分组。在逻辑分组里，你以一定间隔包括了所有数据，根据相对于当前排序键定位的子集来定义。例如，无论是否将其定义为当前月的连续相加，你都创建了相同的分组：

（1）前两行通过 ORDER BY 子句被定义；

（2）任何包含某一月的行不小于两个月以前。

对于连续数据和那些想根据顺序文件的程序员来说，物理分组工作得不错。物理分组比逻辑分组适用于更多的数据类型，这是因为它不需要对值进行操作。

对于在排序中具有间隔或不规则的数据和仔细思考 SQL 谓词的程序员来说，逻辑分组表现得更好。逻辑分组只在你能对值进行算术运算的前提下工作，如数量和日期。

物理分组基于在一个分区内聚合固定数量的行，根据它们相对于行的位置做函数运算。一个通用的格式为：

```
OVER (RANGE BETWEEN <bound_1> AND <bound_2>)
```

窗口的开始<bound_1>可为：

```
UNBOUNDED PRECEDING
<unsigned constant> PRECEDING
<unsigned constant> FOLLOWING
CURRENT ROW
```

意义是显而易见的。UNBOUNDED PRECEDING 包括整个分区，该分区的行在排序顺序的当前行之前。编号的位移通过对行计数来完成。

窗口的结束<bound_2>可为：

```
UNBOUNDED FOLLOWING
<unsigned constant> PRECEDING
```

```
<unsigned constant> FOLLOWING
CURRENT ROW
```

UNBOUNDED FOLLOWING选项包含了整个分区，该分区的行在排序顺序的当前行之后。例如，可以包含整个分区：

```
OVER (RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING)
```

缩写的ROWS选项只包含之前的行。例如，下面是运行过程中的累加求和：

```
SELECT SUM(x)
 OVER (ROWS UNBOUNDED PRECEDING) AS running_total
 FROM Foobar;
```

11.8.2 NTILE(n)

NTILE(n)将一个集合切分为相等的组，每组大约*n*行。这经常包括厂商扩展和关于桶的规则，所以要小心使用。例如：

```
NTILE(3) OVER (ORDER BY x)
x NTILE
================
1 1
1 1
2 1
2 1
3 2
3 2
3 2
3 3
3 3
3 3
```

SQL引擎尝试将每组分为一样大小，但这并不总是可行的。那么目标就定为让它们只差一行。如果在查询中 n 大于行数，那么 NTILE(n)其实是个高效的以容量1进行分组的ROW_NUMBER()函数。

显然，如果使用NTILE(100)，会得到百分位数，但是至少需要结果集中的100行。一个去除离群值（一个在其他数据值范围以外的值）的技巧是使用NTILE(200)，然后通过舍弃第1个和第200个桶来去除正态分布两边的0.5%。

11.8.3 嵌套的OLAP函数

这一点将会使老的SQL程序员困惑。这些OLAP扩展是标量函数，而不是合计。不能在标准SQL中嵌套合计，因为那是毫无意义的。考虑下面这个例子：

```
SELECT customer_id, SUM(SUM(purchase_amt)) --error
 FROM Sales
 GROUP BY customer_id;
```

每个消费者通过里面的 SUM() 得到自己的购买行为合计，这是一个针对分组的数字。如果成功了，最外层的 SUM() 就会对单个数字进行合计。但是，可以这么写：

```
SUM(SUM(purchase_amt)OVER (PARTITION BY depart_nbr))
```

这个例子中，先计算每个部门的购买合计，然后再加总。

11.8.4 查询样例

可能行编号最常用于在前端显示。这并不是好事，因为展示应该是在前端完成而不是在数据库中完成。但是，在这里：

```
SELECT invoice_nbr,
 ROW_NUMBER()
 OVER (ORDER BY invoice_nbr) AS line_nbr,
 FROM Invoices
 ORDER BY invoice_nbr;
```

现在让我们尝试一些更像报表的工作。列出公司里工资最高的 5 个人：

```
SELECT emp_nbr, last_name, sal_tot, sal_rank
 FROM (SELECT emp_nbr, last_name, (salary + bonus)
 RANK()
 OVER (ORDER BY (salary + bonus) DESC)
 FROM Personnel)
 AS X(emp_nbr, last_name, sal_tot, sal_rank)
 WHERE sal_rank < 6;
```

衍生的表 X 计算排名，然后包含的查询裁剪至前五。

有一个销售机会和经销商的表，我们想基于他们的 ZIP 码进行匹配。每个经销商有一个优先级，每个销售机会都有一个被接收的日期。最高优先级的经销商得到最早的销售机会。

```
(SELECT lead_id,
 ROW_NUMBER()
 OVER (PARTITION BY zip_code
 ORDER BY lead_date)
 AS lead_link
 FROM Leads) AS L
FULL OUTER JOIN
(SELECT dealer_id,
 ROW_NUMBER()
 OVER (PARTITION BY zip_code
```

```
 ORDER BY dealer_priority DESC)
 AS dealer_link
 FROM Dealers) AS D
ON D.dealer_link = L.lead_link
AND D.zip_code = L.zip_code;
```

可以在 ORDER BY 的列表中增加更多的条件，或者创建一个有多参数评分标准的查找表。

11.9　稀疏的数据立方体

在考虑报表数据时，数据立方体的概念很容易让人得到一个图形图像。想想简单的电子表格，列保存了发货日期，行保存了那天发货的产品。显然，如果公司有 10 000 件商品和 5 年的历史，这将是一个巨大的电子表格（实际上有 3 652 500 个单元格）。

多数单元格都是空的。但是，空、0 和 NULL 之间有一点微妙的区别。“空”是电子表格的术语，它意味着这个单元格存在是因为它是在电子表格被创建时由行列范围生成的。“0”是一个数值，它需要存在于单元格中，并且它不能作为除数——它是一个真正的数字。“NULL”是一个 SQL 的概念，它表示缺失或者未知值的占位。记住，NULL 可以通过简单的计算传播并且在 SQL 中需要存储空间，它们和空的单元格不一样。

想象一下，我们在 2001 年 10 月 23 日之前没有运送过 iPod，这是因为那时 iPod 还没有发布。在数据立方体里这个单元格就不存在。在 2001 年 10 月，我们有红丝绒、低腰裤、喇叭裤的库存，但我们没有售出一件（从 1976 年每月都这样），这是一个 0 值。最后，还没有人报告 2006 年 3 月 25 日 iPod 的运送情况，但我们就知道这将是个热销品，并且我们将开始销售。现在我们为 iPod 增加一列，历史记录的空单元格就出现了。

这时候，需要决定如何处理这些案例。我推荐忽略在任何销售历史报表里不存在的 iPod。数据立方体工具应该能够区别这 3 种例子的不同。但是，红丝绒、低腰裤和喇叭裤没有长长的销售历史记录（也就是说运送量为 0）是一个很重要的信息——嘿，迪斯科时代已经过去了，你需要清理库存。

NULL 同样也是信息，但是更恰当的说法是它一个数据缺失的标志。这是比较棘手的，因为你需要想办法去处理这些缺失数据。

你能估计某个值并加以利用吗？如果 iPod 销售情况以每月 p%的速率稳定增长 m 个月，我能假定这个趋势会持续下去吗？或者应该使用一个中位数或均值吗？或应该充分利用数据中的价值吗？

11.9.1　数据立方体

将电子表格模型进行扩展，从二维扩展至三维、四维、五维等。人类对于超过三维的图

像有严重的问题。我们居住的宇宙和看见的事物都受限于三维。

大量的单元格都会是空的，至关重要的是数据立方体的存储是用稀疏矩阵来实现的。这意味着空的单元格并没有占用物理内存，但是它们可能被物化在引擎中。图 11-1 所示的是一个数据立方体（源、路线、时间三维）的图解。我将会基于这 3 个坐标轴来解释这些分层。

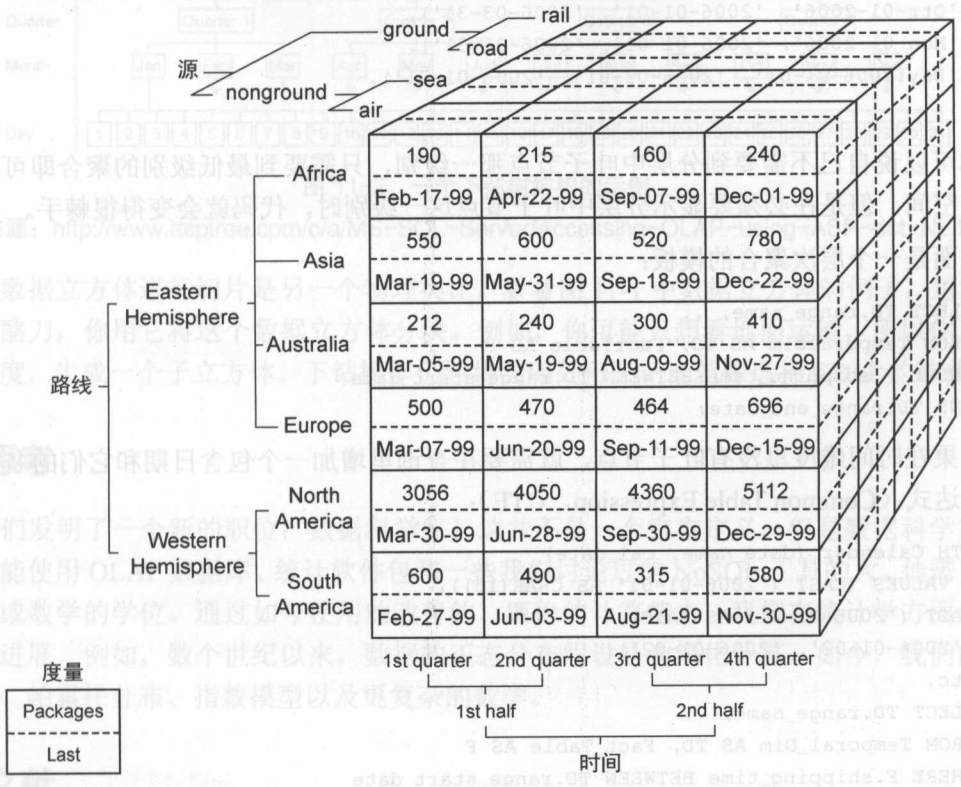


图 11-1 一个数据立方体（源，路线，时间）

（来源：http://www.aspfree.com/c/a/MS-SQL-Server/Accessing-OLAP-using-ASP-dot-NET/）

11.9.2 维度分层

列出数据立方体里的所有单元格是没什么用的。我们想要知道聚合（汇总）信息。我们想要知道 iPod 销售在增长，知道我们需要摆脱掉这些喇叭裤，知道最多的业务来自于北美，等等。

聚合意味着我们需要基于维度的分层结构。下面是一个关于时间维度的例子。如果必须编写自己的代码，那么通过嵌套集合模型在 SQL 中实现分层是很容易实现的：

```
CREATE TABLE TemporalDim
(range_name CHAR(15) NOT NULL PRIMARY KEY,
```

```
range_start_date DATE NOT NULL,
range_end_date DATE NOT NULL,
 CHECK (range_start_date<range_end_date));

INSERT INTO TemporalDim
VALUES ('Year2006', '2006-01-01', '2006-12-31'),
 ('Qtr-01-2006', '2006-01-01', '2006-03-31'),
 ('Mon-01-2006', '2006-01-01', '2006-01-31'),
 ('Day:2006-01-01', '2006-01-01', '2006-01-01'),
 ..;
```

你可以说自己不需要到分层中叶子节点那一级别，只需要到最低级别的聚合即可。这样会节省空间，但是在必须要显示分层中叶子节点这一级别时，代码就会变得很棘手。

下面是一个层次聚合的模板：

```
SELECT TD.range_name, ..
 FROM TemporalDim AS TD, FactTable AS F
 WHERE F.shipping_time BETWEEN TD.range_start_date
 AND TD.range_end_date;
```

如果在时间维度里没有叶子节点，就需要在查询里增加一个包含日期和它们的名字的公共表表达式（Common Table Expression，CTE）：

```
WITH Calendar (date_name, cal_date)
AS VALUES (CAST ('2006-01-01' AS CHAR(15)),
 CAST ('2006-01-01' AS DATE),
 ('2006-01-02', '2006-01-02'),
 etc.
SELECT TD.range_name, ..
 FROM Temporal_Dim AS TD, Fact_Table AS F
 WHERE F.shipping_time BETWEEN TD.range_start_date
 AND TD.range_end_date
UNION ALL
SELECT Calendar.date_name, ..
 FROM Calendar AS C, FactTable AS F
 WHERE C.cal_date = F.shipping_time;
```

日历表是用来进行 OLTP 方面的其他查询，所以应该已将其保存在至少一个数据库中。你可能还会发现你的数据立方体工具自动返回你请求的叶子节点级别的数据。

11.9.3　下钻和切片

许多 OLAP 的用户界面工具都有一个下拉菜单的展示，里面包含每个维度，让你可以选择报表中的聚合级别。就好像昂贵的环绕立体音响设备前面的滑动开关是用来设定音乐是如

何播放的，而不是用来选择播放 CD 里面的哪首歌曲的。这称为下钻，因为是在分层的最高级别开始并沿着树形结构下行。图 11-2 是微软报表工具的界面的例子。

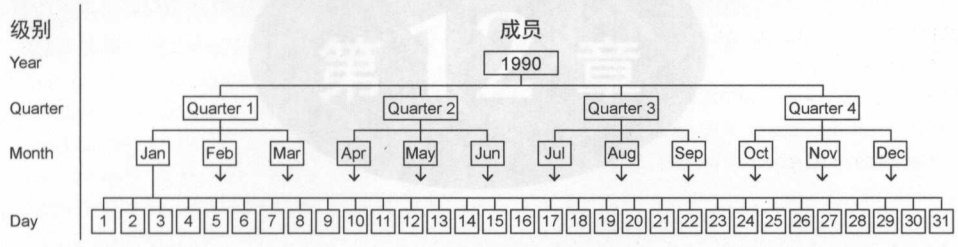


图 11-2 一个下钻树结构的示例

（来源：http://www.aspfree.com/c/a/MS-SQL-Server/Accessing-OLAP-using-ASP-dot-NET/）

对数据立方体进行切片是另一个物理类比。看看图 11-1 中数据立方体的例子，想象你有一把奶酪刀，你用它将这个数据立方体分块。例如，你可能只想看地面运输，所以你只切开那个维度，生成一个子立方体。下钻操作仍然将会随处可见。这就像选择 CD 上的歌曲一样。

总结思考

我们发明了一个新的职位：数据科学家！这并不是一个官方定义，但是数据科学家应该是一个能使用 OLAP 数据库、统计软件包和一些我们讨论过的 NoSQL 工具的人。他需要一个统计学或数学的学位。通过如今使用的改善的、廉价的计算能力，我们在统计学方面也取得了很多进展。例如，数个世纪以来，数据的正态分布假设是为简化计算。如今，我们能使用重采样、帕累托分布、指数模型以及更复杂的数学。

参考文献

Codd, E. F. (1993). *Providing OLAP to User-Analysts: An IT Mandate*.Available at, http://www.minet. uni-jena.de/dbis/lehre/ss2005/sem_dwh/lit/Cod93.pdf.

Zemke, F., Kulkarni, K., Witkowski, A., & Lyle, B. (1999). *Introduction to OLAP functions* (ISO/IEC JTC1/ SC32 WG3:YGJ-068 ANSI NCITS H2-99-154r2). ftp://ftp.iks-jena.de/mitarb/lutz/standards/sql/OLAP-99-154r2.pdf.

多值数据库或 NFNF 数据库

简介

RDBMS 是基于第一范式的，它假设了数据是以标量值的形式保存在行中的列里，而且这些记录有着相同的结构。多值模型允许将表内嵌到列中。它们有一个不被 SQL 程序员所熟知的小众市场。这种数据模型有一种代数理论，它听起来就像关系型模型。

2013 年，大多数数据库程序员只用过 SQL。他们并没有随着文件系统、COBOL、任何老式的网络数据库或层次数据库一起成长。这些产品仍然存在并且运行了很多商业应用。但是我们也看到了传统层次顺序记录和面向集合数据模型的互相混合。让我们从某些历史开始。

12.1 嵌套文件结构

平面文件的字段都是标量值并且记录有相同的结构。当在 20 世纪 70 年代关系型模型第一次出现时，开发人员误认为表是平面文件。他们弄错了数学运算和集合的概念。文件按照顺序从左读到右，一条记录接一条记录；一个表就像一个集合，是工作单元。文件是独立的，而表是模式的一部分，而且与模式的其余部分存在关系。坦白讲，很多开发者还是没有弄清这些概念。

但是平面文件是最容易的起点。接下来的最复杂的文件结构有多样化记录。由于文件是从左读到右，它可以告诉计算机预期的数据“下游”是什么。在 COBOL 中，我们使用 OCCURS 和 OCCURS DEPENDING ON 子句。

假设大多数读者不知道 COBOL。我会很快地说明 COBOL 的 DATA DIVISION 就像 SQL

中的 DDL，但是以字符串形式保存的数据有一个图片（PIC）子句，该子句展示了它们的显示格式。在 COBOL 中，显示和存储的格式相同。记录组成了字段的层次结构，并且嵌套等级如每个声明开始的整型数字所示（数字随着深度增加而增加，惯例是步长为 5）。假设你想要存储你这一年的月度销售数据，那么可以定义 12 个字段来对应 12 个月，具体如下：

```
05 MONTHLY-SALES-1 PIC S9(5)V99.
 05 MONTHLY-SALES-2 PIC S9(5)V99.
 05 MONTHLY-SALES-3 PIC S9(5)V99.
 ...
05 MONTHLY-SALES-12 PIC S9(5)V99.
```

破折号与 SQL 的下划线类似，句点和 SQL 中的分号类似，PIC 子句告诉我们每个销售数量有一个符号，5 位数字表示美元，2 位数字表示美分。你可以一次性指定字段，然后使用简单的 OCCURS 子句重复 12 次，具体如下：

```
05 MONTHLY-SALES OCCURS 12 TIMES PIC S9(5)V99.
```

在 COBOL 通过下标引用独立字段，如 MONTHLY-SALES(1)。OCCURS 也能用于组级别，这是它最有用的应用了。例如，发票的所有 25 行（75 个字段）能在这组里保存：

```
05 LINE-ITEMS OCCURS 25 TIMES.
10 ITEM-QUANTITY PIC 9999.
10 ITEM-DESCRIPTION PIC X(30).
10 UNIT-PRICE PIC S9(5)V99.
```

注意，OCCURS 被列在了组级别中，所以整个组出现了 25 次。

还可以使用嵌套的 OCCURS。假设存储了 10 个产品，我想保存过去 12 月每个产品每月的销量：

```
01 INVENTORY-RECORD.
 05 INVENTORY-ITEM OCCURS 10 TIMES.
 10 MONTHLY-SALES OCCURS 12 TIMES PIC 999.
```

本例中，INVENTORY-ITEM 是一个仅由 MONTHLY-SALES 组成的组，它要为每一个库存商品重复发生 12 次。这一共带来了 12×10 个字段的数组。这个记录唯一包含的信息是 120 个月度销售数据——10 个商品，每个商品都销售了 12 个月。

注意，OCCURS 定义了一个已知大小的数组。但是由于 COBOL 是一种文件系统语言，所以它从左到右地从记录中读取字段。由于不存在 NULL 值，所以想插入未知的值需要一些编码技巧。COBOL 有 OCCURS DEPENDING ON 选项。计算机读取一个整型控制字段，然后在运行时希望找到许多子记录的出现。是的，这会搞得杂乱和复杂，但是来看看这个简单的病例来得到一个感性的认识。

```
01 PATIENT-TREATMENTS.
05 PATIENT-NAME PIC X(30).
05 PATIENT-NUMBER PIC 9(9).
05 TREATMENT-COUNT PIC 99 COMP-3.
05 TREATMENT-HISTORY OCCURS 0 TO 50 TIMES
DEPENDING ON TREATMENT-COUNT
INDEXED BY TREATMENT-POINTER.
10 TREATMENT-DATE.
15 TREATMENT-YEAR PIC 9999.
15 TREATMENT-MONTH PIC 99.
15 TREATMENT-DAY PIC 99.
10 TREATING-PHYSICIAN-NAME PIC X(30).
10 TREATMENT-CODE PIC 999.
```

在应用中必须通过处理 TREATMENT-COUNT 来正确地形容 TREATMENT-HISTORY 子记录。这里，不会解释 COMP-3（一种用来计算的数据类型）或者 INDEXED BY 子句（数组索引），因为它们不是本书的重点。

重点是，在关系型模型以前我们就已经在思考的嵌套结构的数组数据。我们只是还没有将计算和表示层的数据分开，也还没有找到计算的抽象模型。

12.2 多值系统

当迷你电脑出现后，与当前的硬件和软件相比，它们能力有限。与带有应用语言的数据库融合的应用开发系统是用户们的最爱。用一个工具就能构建完整应用！

类似的、最成功的工具之一是 Pick。它于 1965 年由 Don Nelson 和 Dick Pick 在 TRW 公司开发，是用来供美军控制 Cheyenne 直升机零部件库存的，它的前身是 IBM 的 System/360 上的广义信息检索语言系统（Generalized Information Retrieval Language System，GIRLS）。

这时关系型模型并不存在。事实上，此时没有任何的数据理论。Pick 的文件结构由变长字符串组成。这并不是 COBOL 使用的模型。在 Pick 中，将记录称为条目，字段称为属性，子字段称为值或子值（所以才有今天多值数据库这个术语）。所有元素都是可变长度，字段和值被特殊分隔符所标记，所以任何文件、记录或者字段都可能包含任意数量的低级实体的条目。

结果，Pick 的条目（记录）可能是完整的实体（例如，完整的客户订单）而非带有表的 RDBMS 模型，这里的表是所有订单头的集合，订单头关联到了所有客户订单细节的集合，订单细节又关联到了库存等。

Pick 系统是为虚拟机编写的，并且它包含了类 Unix 的目录分层、子目录和被散列到桶里

以便扫描的记录文件。数据字典将系统结合在了一起。它也有命令行语言，所以它是独立的。

这使得移植 Pick 到其他平台很容易。很快它就被许多分销商授权，所以 Pick 变成了一个 Pick/BASIC 实现的多值数据库家族的通用名字。Dick Pick 成立了 Pick & Associates，随后更名为 Pick System，接着又更名为 Raining Data，2011 年，再次更名为 TigerLogic。他将 Pick 授权给各种各样生产 Pick 不同方言的厂商和供应商。TigerLogic 售出的方言有我们熟知的 D3、mvBase 和 mvEnterprise。那些先前由 IBM 售出属于 U2 系列的称为 UniData 和 UniVerse。Rocket *Software 在 2010 年购买了 IBM 的 U2 产品线。*

Pick 在大部分的微机、个人电脑和大型计算机上运行，截至 2013 年仍被使用。这是一个 Pick 产品家族的简短列表：

- UniVerse（基于 Unix）；
- UniData（基于 Unix）；
- D3；
- jBASE；
- ARev；
- Advanced Pick；
- mvBase；
- mvEnterprise；
- R83。

如果你记得 Ashton-Tate 出品的 dBase，它作为 PC 上第一个流行的数据库产品，你就能比较它和那条演变成 Xbase 产品家族的开发路线了。

Pick 是在 1973 年由 Microdata Corporation（和它们的英国分销商 CMC）作为 Reality Operating System 首次商业性质地发布，现在由 Northgate Information Solutions 提供。Microdata 的实现版本增加了名为 Pick/BASIC 的 BASIC 语言（见 Data/BASIC）。这就变成了事实上的 Pick 开发语言，这是因为它扩展了智能终端接口以及数据库操作。

最后，像所有的 NoSQL 产品那样，他们增加了一种名为 ENGLISH 的 SQL 风格语言（后面改为 ACCESS，不要和微软同名的数据库系统混淆）来检索和生成报表。ENGLISH 最初不能更新，但是后面增加了 `REFORMAT` 命令用来批量更新。ENGLISH 没有连接或者其他关系型操作符。实际上，你在 Pick 里通过用某个字段的数据字典重定义来“预连接”表，这会在另一个文件里执行查找计算。在应用代码里，必须实现数据的完整性。

虽然特有的变化和增强不断出现，但是核心产品仍保持不变。Pick 主要是用来处理业务数据，这是因为它的数据模型很契合办公室工作中使用的文件和表，并且它有一个特别活跃的用户社区。

12.3 NFNF 数据库

程序语言都有一个正式的基础，例如，FORTRAN 是基于代数，LISP 是基于列表处理等。直到 Codd 博士提出他的关系代数理论之前，数据和数据库都没有“学术合法性”。关系代数理论有着学者喜欢的一切——一系列数学表达式，包括那些让排字工人抓狂的新奇表达式。多亏了 Armstrong，它同样也有公理。

直接产生的结果是大量使用 Codd 博士的关系代数理论的论文迅速涌现。但是现代学术的下一步是改变或舍弃某一个公理来看看是否还能有一个一致的正式系统。在几何学中，改变平行公理（平行线永不相交）。例如，改变后的公理为两条平行线（大圆弧）在球体表面的两个点相交。球体是真实存在的。我们就能使用真实世界中的模型来测试这个新的几何理论。

由于第一范式（1NF）是 RDBMS 的基础，它是被学者首先提出的。来检验一下它是否适用于真实的多值数据库。大多数学术工作由 IBM 的 Jaeschke 和 Schek，以及得克萨斯大学奥斯丁分校的 Roth、Korth 和 Silberschatz 完成。它们对关系代数和微积分增加了新的操作来处理“嵌套的关系”，然而这种关系保留了关系型模型的抽象的面向对象特性。1NF 不擅长处理有着复杂内部结构的数据，如计算机辅助设计和制造（CAD/CAM）。这些应用必须处理结构化的实体，然而遵循第一范式的表只允许不可再分的属性值。

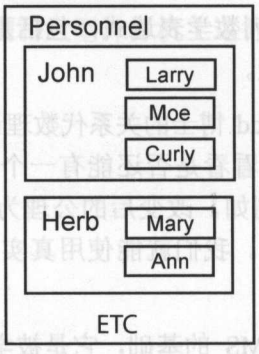
NFNF（Nonfirst normal form，反第一范式）数据库允许表中的列存储嵌套结构，并且打破了在已知域中的列只包含标量值的原则。除了 NFNF 之外，这些数据库在文献中被称为 2NF、NF^2 和¬NF。由于它们不是 ANSI/ISO 标准的一部分，你会发现在操作上会有许多特有的实现和学术符号。

```
CREATE TABLE Personnel
(emp_name VARCHAR(20) NOT NULL PRIMARY KEY,
..);

CREATE TABLE Dependents
(dependent_name VARCHAR(20) NOT NULL PRIMARY KEY,
 emp_name VARCHAR(20) NOT NULL
 REFERENCES Personnel(emp_name)--- DRI actions
 ON UPDATE CASCADE
 ON DELETE CASCADE,
..);
```

但是，在NFNF模式里，亲属以表的类型被保存在列里，具体如下：

```
CREATE NF TABLE Personnel
(emp_name VARCHAR(20) NOT NULL PRIMARY KEY,
 Dependents TABLE
 (dependent_name VARCHAR(20) NOT NULL PRIMARY KEY,
 Emp_name VARCHAR(20) NOT NULL,
 ..),
 ..);
```



可以以合适的方式扩展基本操作符集合UNION、INTERSECTION、DIFFERENCE和子集。扩展关系型操作对于PROJECTION和SELECTION来说也不难。JOIN操作有一点儿困难，但是如果将关系代数限制为自然连接或等值连接，那么就会容易些。最重要的特性是当这些扩展的关系型操作被用于平面表时，它们与最初的关系型操作表现得一样。

为了将这个NFNF表转换为1NF模式，你要使用UNNEST操作符。在本例中非嵌套处理，会将Dependents移到自己表中，并将其移出Personnel，尽管对于NEST来说，UNNEST是一个数学上的反转。让我们从一个简单的、抽象的嵌套表开始：

G1			
F1	F2	G2	
		F3	F4
X	Y	X	Y
		Y	X
X	Y	Y	Z
		Z	X

UNNEST(<subtable>)操作将子表平铺到嵌套中的上一级：

G1			
F1	F2	F3	F4
X	Y	X	Y
X	Y	Y	X
X	Y	Y	Z
X	Y	Z	X

嵌套操作需要新表名和它的列作为参数。以下是 SQL 声明的扩展：

```
NEST (G1, G2(F3, F4))
```

G1			
F1	F2	G2	
		F3	F4
X	Y	X	Y
		Y	X
		Y	Z
		Z	X

当想要通过“再嵌套”这一步骤来返回到原始表时，它不会成功。

对于如何进行嵌套还有一个问题。在第一个 `Personnel` 例子中，我们将亲属放到了 Personnel 表中。孩子是弱实体，他们必须依附于父母（强实体）存在。但是我们可以将父母嵌套到亲属里。问题来了，NEST() 不能交换。如果忘记了高中代数，这里来回顾下，当(A○B)=(B○A)，操作就是可交换的。从一个简单的平面文件开始：

G1			
F1	F2	F3	F4
X	Y	X	X
X	Y	X	Y
X	Y	X	Z
X	Y	Y	X
X	Y	Z	X

现在，进行两次嵌套来生成由 F3 列组成的子表 G2 和由 F4 列组成的子表 G3。首先，以这个顺序：

```
NEST(NEST (G1, G2(F3)) , G3(F4))
```

G1			
F1	F2	G2	G3
		F3	F4
X	Y	X	X
		Y	
		Z	
X	Y	X	Y
			Z

现在以相反的顺序：

```
NEST(NEST (G1, G3(F3)) , G2(F4))
```

G1			
F1	F2	G2	G3
		F3	F4
X	Y	X	X
			Y
			Z
X	Y	Y	X
		Z	

下一个问题是如何处理缺失的数据。在 Personnel 表的例子中，如果 Herb 的女儿 Mary 是乳糖不耐受症患者并且没有喜欢的冰激凌口味，那该怎么办呢？通常的 NFNF 模型需要明确的标记而不是通用的缺失值。

另一个对于操作的限制要求来说是客观存在的，它被划分范式（PNF）所包含。该范式不能有空的子表并且操作必须是可逆的。更正式的说法是，PNF 关系的原子属性集是关系的超键，且关系上的元组的任何非原子组成部分也都是满足 PNF 的。

12.4　现有的表值扩展

现有的 SQL 产品都新增了一些 NFNF 扩展，但是它们并没有被很好优化。即使有 ANSI/ISO 标准，语法通常还是各自的方言。

12.4.1　Microsoft SQL Server

Microsoft SQL Server 2008 新增了一些表值参数和用户自定义数据类型。用户可以通过语法将表声明为数据类型，那么就能在存储过程中使用这个类型名来定义本地变量或参数。根据微软的说法，对于多达 1000 行左右的记录表值参数是高效的选择。语法很直白：

```
CREATE TYPE Payments
AS TABLE (account_nbr CHAR(5) NOT NULL,
 payment_amt DECIMAL(12,2));
```

这实际上不是一种 NFNF 的实现，它更多的是一种有限制的简化版。不能在数据表声明里使用用户定义数据类型！T-SQL 方言为本地变量使用了@前缀，可以为标量和表变量，所以想在会话中得到一个本地的临时表，必须这么写：

```
DECLARE @atm_money Payments;
```

12.4.2 Oracle 扩展

Oracle 拥有比 T-SQL 更强大的实现，但是如果展平嵌套表，就无法进行优化。语法使用了 DDL 中的 Oracle 对象类型。产品里还有其他集合类型，但是这里只需为这个简单的例子创建一个 Address_Type 类型。这条 DDL 命令将给我们一个 Address_Type 和 Address_Book 表。Address_Book 是包含 Address_Type 类型的表，与我们刚才在 T-SQL 中看到的语法模型很像：

```
(addr_line VARCHAR2(35),
city_name VARCHAR2(25),
 state_code CHAR2(2),
 zip_code CHAR2(5));

CREATE TYPE Address_Book
AS TABLE OF Address_Type;
```

所以，想创建一个表，只需指定列名（本例中的 emp_addresses）和新创建的类型（Address_Book）：

```
CREATE TABLE Personnel
(emp_name VARCHAR2(25),
 emp_addresses Address_Book);
```

为了使用这张表和嵌套表，首先将地址对象放进一张名为 Personnel 的基本表中。嵌套表目前是空的，所以我们在 INSERT INTO 语句里使用关键列名来将其填满。为了生成一个结果集可以利用表函数：

```
INSERT INTO Personnel(name, emp_addresses)
 VALUES ('Fred Flintstone', Address_Book());

INSERT INTO TABLE
(SELECT emp_addresses
 FROM Personnel
```

```
 WHERE emp_name = 'Fred Flintstone')
 VALUES ('123 Main', 'Bedrock', 'TX', '78787');

INSERT INTO TABLE
(SELECT emp_addresses
 FROM Personnel
 WHERE emp_name = 'Fred Flintstone')
 VALUES ('12 Lava Ln', 'Slag Town', 'CA', '98989');

INSERT INTO TABLE
(SELECT emp_addresses
 FROM Personnel
 WHERE emp_name = 'Fred Flintstone')
 VALUES ('77 Cave Ct', 'Pre-York', 'NY', '12121');
```

为了查看结果，可以通过一个简单的查询来展平嵌套结构。emp_addresses 表被看成一个派生表表达式，但是这里有一个隐含的连接条件：

```
SELECT T1.emp_name, T2.*
FROM Personnel AS T1,
TABLE(T1.emp_addresses) AS T2;
```

Fred Flintstone	123 Main St	Bedrock	TX	78787
Fred Flintstone	12 Lava Ln	Slag Town	CA	98989
Fred Flintstone	77 Cave Ct	Pre-York	NY	33333

总结思考

对于这个家族的产品，多值数据库可能是一个比 NFNF 数据库更好的名字了。根据事物的本质来定义它是什么比否定一些东西要好些。大多数数据库程序员学习过关系型模型，但是除了一些小众人群，没有人知道其他模型，如这里讨论的多值模型。

层次数据库系统和网络数据库系统

简介

IMS 和 IDMS 是最重要的前关系型技术，直到今天仍被广泛使用。事实上，有一个好机会，IMS 比 SQL 数据库保存有更多的商业数据。这些产品仍然在银行业、保险业和大型机上的大型商业应用里“繁忙工作”，并且它们使用 COBOL。它们对于那种变化不大而又需要移动大量数据的场景非常适合。由于如此多的数据仍然存在于 IMS 和 IDBS 中，所以你必须至少了解分层和网络数据库系统的基本知识，以便从中得到数据再用 NoSQL 工具保存。

我会假设本书的大多数读者只用过 SQL。如果你在大学的数据库课程中听过层次或者网络数据库系统，然后逐渐淡忘了。从某些方面来说，这太糟了。它有助于了解早期的工具是如何工作的，这样你就能明白新的工具是如何从老的工具发展而来的。

13.1 数据库类型

经典的数据库类型有网络、关系型和层次。关系型模型与 E.F.Codd 博士相关，其他两个模型与 Charles Bachman 相关，他于 20 世纪 50 年代在陶氏化学作出了开创性的工作，然后于 20 世纪 60 年代在通用电气开发出了集成数据存储（IDS），这是最初的数据库管理系统之一。网络和层次模型被称为导航数据库，这是因为数据访问的思维模式就像一个读者沿着路径移动来获取数据。事实上，当 Bachman 在 1973 年由于在数据库技术方面的杰出贡献而被授予 ACM 图灵奖时，他就是这么形容它的。

IMS 不是唯一的导航数据库，但是是最流行的。Cincom 公司的 TOTAL 基于主记录，主

记录通过指针链找到一个或多个从记录集合。不久后，IDMS 和其他产品开始推广这种导航模型。

定义 COBOL 的委员会——CODASYL，为这个导航型模型想出了一个标准。最后，ANSI X3H2 数据库标准委员会通过了 CODASYL 模型，将其稍加正规化，并推出了 NDL 语言标准。但是，在那个时候，SQL 已经成为了 ANSI X3H2 数据库委员会的主要工作，没有人真的关心 NDL，这个标准就这么失效了。

IBM 的 IMS 是最流行的层次数据库管理系统，如今仍被广泛使用。它稳定、定义明确、可扩展并且速度很快。IMS 软件环境可以被分为 5 个部分：

- 数据库；
- 数据语言 I（DL/I）；
- DL/I 控制块；
- 数据通信组件（IMS TM）；
- 应用程序。

13.2　数据库历史

在 DBMS 发展之前，数据是被保存在单独的文件中。系统里，每个文件以顺序或索引的格式被存储在一个单独的数据集里。为了从文件中检索数据，应用必须打开文件并且读到所需数据的位置。如果数据分散在大量的文件里，数据访问就需要频繁地打开和关闭文件，这样会产生额外的输入/输出（I/O）和处理开销。

为了减少被应用访问的文件数，程序员经常在许多文件中保存相同的数据。这样就会产生冗余数据和跨文件保证更新一致性的相关问题。为了保证数据一致性，特殊的跨文件更新程序必须按照计划将原始文件更新。

数据库系统的概念解决了在一个文件系统中遇到的许多的关于数据完整性和数据备份的问题。一个设计良好的数据库在一处地方只存储一次数据，并将其对所有应用程序和用户开放。同时，数据库通过限制对数据的访问来提供安全性保证。用户读、写、更新、插入或删除数据的能力就会被限制。与平面文件的集合相比，在单个数据库中数据也更容易备份和恢复。

数据库结构为数据检索提供了多种策略。应用程序能够顺序检索数据或（通过特定的访问方法）直接找到所需数据，减少 I/O 并提升数据检索的速度。最后，对数据库某部分的更新操作对其他应用是立即可用的。因为数据只存在于一处，所以数据完整性更容易保证。

如今存在的 IMS 数据库管理系统体现了层次数据库多年的发展和提升的演变结果。全世

界大量商业和政府设备都在使用 IMS。IMS 公认为各种应用提供卓越的性能支持，并且公认在有着中等到大量的数据和事务的数据库下表现良好。

13.2.1 DL/I

由于它们是通过使用 DL/I 被实现和被访问的，IMS 数据库有时也被称为 DL/I 数据库。DL/I 是一种命令级语言，而不是一个数据库管理系统。DL/I 用于批处理程序和在线程序从而访问数据库里的数据。

应用程序使用 DL/I 来获取需要的数据。DL/I 使用系统访问方法，如虚拟存储访问法（VSAM），来操作从数据库输入和输出的物理传输过程。IMS 数据库经常在它们设计的访问方法里被提到。例如 HDAM（分层直接存取法）、HIDAM（分层索引直接访问法）、PHDAM（分区的 HDAM）、PHIDAM（分区的 HIDAM）、HISAM（分层索引顺序访问法）和 SHISAM（简单 HISAM）。这些都是从它们的大型数据库产品里得到的 IBM 术语，这里不会讨论它们。

IBM 为 9 种类型的访问方法制定了规则，你可以根据它们中的任意一个方法设计数据库。另一方面，SQL 程序员通常被与他们的数据库引擎使用的访问方法隔离开来。我们不会担心这个级别被调用的访问方法的细节。

13.2.2 控制块

创建 IMS 数据库时，必须定义数据库的结构、数据被访问和被应用程序使用的方式。这些规定是在两个控制块提供的参数中被定义，这两个控制块也称为 DL/I 控制块：

- 数据库描述（DBD）；
- 程序规范块（PSB）。

大体上，DBD 描述了数据库的物理结构，PSB 描述了数据库对特定应用程序的可见性。PSB 指定应用程序可以访问数据库的部分和可以对数据执行的函数。DBD 和 PSB 的信息被合并到第三个控制块——应用控制块（ACB）。对于在线处理而言 ACB 是必须的，但是对于批处理来说 ACB 是可选的。

13.2.3 数据通信

IMS 事务管理（IMS Transaction Manager，IMS TM）是在在线、实时环境下提供访问数据库的一系列独立的许可程序的集合。没有 TM 组件，在 IMS 数据库里只能以批处理模式来处理数据。

13.2.4 应用程序

如果数据位于不能被访问的数据库，那么数据库中的数据对你来说没有实际用途。通过

使用业务或组织功能里的应用程序才能发挥它的价值。通过 IMS 数据库，应用程序使用嵌入主机语言的 DL/I 调用来访问数据库。IMS 支持批处理应用程序和在线应用程序。IMS 支持以 ADA、Assembler、C、C++、COBOL、PL/I、Pascal、REXX 和 WebSphere Studio Site Developer 第 5 版编写的程序。

13.2.5　层次数据库

在层次数据库中，数据在记录中被分组，被细分为一系列段。想想学校的系数据库，其中的记录是由 Dept、Cource 和 Enroll 这些段组成的。在层次数据库中，数据库的结构被设计为反应逻辑依赖性——某些数据依赖于其他数据的存在。登记是依赖于课程的存在，并且，在这个案例中，课程是依赖于部门的存在，部门可以提供课程。这些在 RDBMS 中被称为强实体和弱实体。

将专业术语从 SQL 的世界切换到 IMS 的世界。IMS 使用记录和字段，并且将每个层次称作数据库。在 SQL 的世界里，行和列可以是虚拟的，有默认值，并且有约束——它们是智能的。记录和字段是物理存在的，并且要依靠应用程序来赋予它们意义——它们是愚蠢的。在 SQL 里，模式或者数据库是相关表的集合，它们可能映射相同数据模型中几个不同的 IMS 层次。换句话说，IMS 数据库更像 SQL 中的一个表。

13.2.6　优势和劣势

在一个层次数据库里，数据关系是通过存储结构定义的。查询的规则是高度结构化的。当查询还没有被高度优化时，与 SQL 数据库相比，正是这些固定的关系能让 IMS 极其快速地访问数据。

层次和关系型系统有它们的优势和劣势。关系型结构使编写专项报表的代码变得相对容易。但是 SQL 查询经常使引擎通过对一个完整的表或一系列表的全表扫描来检索数据。这使得搜索速度变慢，而且加大了处理压力。另外，由于整个数据库必须保持完整的，所以每张表的每一行的每一列都必须产生一个实体，即使这个实体只是一个占位符（如 NULL）实体。

伴有这些层次结构，构造数据需求或片段搜索参数（SSA）会更加复杂。但是一旦构造完成，它们会非常高效，可以直接对需要的数据进行检索。产生的结果是一个极其快速的数据库系统，这个系统可以处理大量的数据事务和并发用户。同样的，也不需要为没有保存的数据创建占位符。如果不需要某个片段的出现，它就不会被创建或者插入。

你需要在 SQL 的简易性、可移植性和灵活性与 IMS 的速度和节约存储空间之间做出权衡。你要为一系列应用调优 IMS 数据库。

13.3 简单的层次数据库

为了展示层次结构是什么样的，本节设计了两个非常简单的数据库用来存储某个学院的课程和学生信息。一个数据库将会存储学院每个部门的信息，另一个数据库将会包含学院每个学生的信息。在层次数据库里，必须尝试将数据组织为一对多的关系。

还需要设计数据库，这样逻辑上依赖于其他数据的数据就会存储在基于数据进行分层依赖的段上。因此，我们设计 Dept 为记录的键段，或记录的根段，这是因为其他数据不能在缺失系的情况下而存在。每个系只列出一次。基于每个系的每门课程来提供数据。有一个段类型 Course，系里每个课程都是该类型。课程名称、描述以及讲师相关的数据被保存在 Course 段的字段里。最后，我们增加另一个段类型——Enroll，它包括了选课学生的 ID。

注意，这里违反了 ISO-11179 命名规则。导航模型是基于数据元素某一时刻的实例，就像磁带或者卡片文件。不是像 RDBMS 一样是基于整个数据集的。

在图 13-1 中，我们创建了第二个数据库，叫 Student。这个数据库包含了学院里所有选课学生的信息。这个数据库会存储 Department 数据库中 Enroll 段的一部分数据。接着，我们会创建一个更大的数据库，用它来消除冗余数据。数据库的设计依据了很多因素。在本例中，我们将会聚焦需要最经常访问的数据。

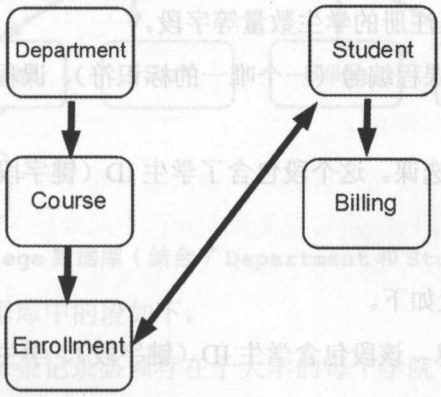


图 13-1 层次数据库的 Department 和 Student 样例

如图 13-1 所示，图中的两个数据库可以以关系型形式构建为 3 个表。注意，当直接转换为 SQL 时，这些表并不总是规范化的。例如：

```
CREATE SCHEMA College;
CREATE TABLE Courses
(course_nbr CHAR(9) NOT NULL PRIMARY KEY,
```

```
    course_title VARCHAR(20) NOT NULL,
    course_description VARCHAR(200) NOT NULL,
    dept_id CHAR(7) NOT NULL
    REFERENCES Departments (dept_id)
    ON UPDATE CASCADE);

CREATE TABLE Students
(student_id CHAR(9) NOT NULL PRIMARY KEY,
 student_name CHAR(35) NOT NULL,
 student_address CHAR(35) NOT NULL,
 major CHAR(10));

CREATE TABLE Departments
(dept_id CHAR(7) NOT NULL PRIMARY KEY,
 dept_name CHAR(15) NOT NULL,
 chairman_name CHAR(35) NOT NULL,
 budget_code CHAR(3) NOT NULL);
```

13.3.1 Department 数据库

Department 数据库中的段如下。

- Dept：每个系的信息。该段包含了系 ID（键字段）、系名称、系主任名字、教师的数量和在系里的课程里注册的学生数量等字段。
- Course：该段包括课程编号（一个唯一的标识符）、课程名称、课程描述和讲师姓名等字段。
- Enroll：学生进行选课。这个段包含了学生 ID（键字段）、学生姓名和年级等字段。

13.3.2 Student 数据库

Student 数据库中的段如下。

- Student：学生信息。该段包含学生 ID（键字段）、学生姓名、地址、专业和完成课程等字段。
- Billing：课程选择的订单信息。该段包含学期、欠费、交纳学费和奖学金申请等字段。

Student 数据库的根（Student）段和 Department 数据库的 Enroll 段之间的双箭头表现出了一种基于某一段中的数据和其他段所需的数据的逻辑关系。这不像在 SQL 中的参考和引用的表结构，它必须由应用程序来执行。

13.3.3 设计考量

在为数据库设计层次结构时，应该分析最终用户的处理需求，这是因为他们将会决定你如何构建数据库。特别的是，你必须考虑数据元素是如何关联以及它们是如何被访问的。

例如，就拿经典的 Parts and Suppliers 数据库来说，层次结构能将零部件划分在供应商的应收账款部门之下，或者将供应商划分在订购部门之下。在 RDBMS 中，在零部件和供应商之间会存在一个关系表通过它们的主键进行关联，它包含了它们的从属关系信息，而不是任意零部件或者供应商的信息。

13.3.4 样例数据库扩展

到这里，你已经学到了足够多的有关数据库设计的内容来扩展最初的样例数据库。可以通过结合 Department 和 Student 数据库来更好地利用学院数据。我们的新 College 数据库如图 13-2 所示。

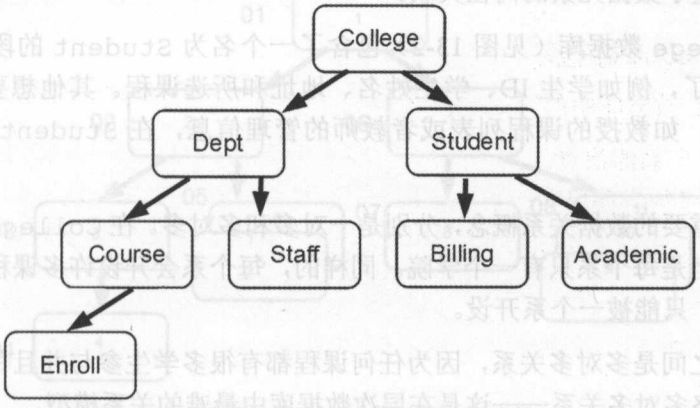


图 13-2 College 数据库（结合了 Department 和 Student 数据库）

在扩展的 College 数据库中的段如下。

- College：根段。每条记录必须存在于大学的每个学院下。键字段是学院 ID，例如 ARTS、ENGR、BUSADM 和 FINEARTS。
- Dept：学院中每个系的信息。该段包含系 ID（键字段）、系名称、主任名称、教师数量和在系里的课程里注册的学生的数量等字段。
- Course：该段包含课程编号（键字段）、课程名称、课程描述和讲师姓名等字段。
- Enroll：在课程里登记的学生列表。有学生 ID（键字段）、学生姓名、当前年级和缺席次数。

- Staff：职员的清单，包含了教授、讲师、助教和文职人员。键字段是雇员编号。还有姓名、地址、电话号码、办公室编号和工作计划字段。
- Student：学生信息。该段包含了学生ID（键字段）、学生姓名、地址、专业和当前选课字段。
- Billing：订单和支付信息。它包含了订单日期（键字段）、学期、订单数额、支付数额、奖学金申请和奖学金可用字段。
- Academic：键字段是年份和学期的结合体。字段包括每学期的平均成绩点数（GPA）、累积的GPA字段和列出完成课程和每学期成绩的足够字段。

13.3.5　数据关系

数据标准化的过程能帮助你将数据拆分为有自然联系的分组，这些分组可以被整体保存在层次数据库的段里。在设计数据库时，将独立的数据元素拆分为基于处理函数服务的组。同时，分组数据基于数据元素的内在关联。

例如，College数据库（见图13-2）包含了一个名为Student的段。围绕学生一些数据被联系起来了，例如学生ID、学生姓名、地址和所选课程。其他想要从College数据库获取的数据，如教授的课程列表或者教师的管理信息，在Student段里不能很好地工作。

这里有两个重要的数据关系概念，分别是一对多和多对多。在College数据库钟，每个学院有很多系，但是每个系只有一个学院。同样的，每个系会开设许多课程，但是一个特定的课程（本例中）只能被一个系开设。

课程和学生之间是多对多关系，因为任何课程都有很多学生参与并且每个学生会选修多门课程。暂时忽略多对多关系——这是在层次数据库中最难的关系模型。

作为层次数据库中的一种依赖关系一对多关系被被构建：多个依赖一个。没有系，就不会开设课程；没有学院，也就不会存在系。

父母和孩子之间的关系仅仅基于层次中段的相对位置，并且一个段可以是其他段的父段，同时也可以是它之上的段的子段。在图13-2中，Enroll是Course的子段，Course是Enroll的父段，而Course也是Dept的子段。Billing和Academic都是Student的子段，而Student也是College的子段，除了College的所有段都依赖于其他段。

如果分析了数据元素，将它们分组到段中，为每个段挑选一个键字段，然后数据库结构，就已经完成了大部分数据库设计了。你可能发现，不管怎样，你选择的设计并不会适用于每一个应用程序。一些应用程序可能需要通过不同于你选择作为键的字段来访问段。或者另外

的应用可能需要将位于两个数据库或者层次结构的段联系起来。IMS 提供了两个非常有用的工具，你可以用它们来解决这些数据需求：二级索引和逻辑关系。

二级索引可以基于不同于根段键字段的字段建立索引。这个字段可以这样被使用：就好像它是基于某个数据元素来访问段的不同于根键的键。

逻辑关系可以在分离的层次中对段建立联系。实际上，逻辑关系创建了*在存储中并不存在的层次结构*。逻辑结构可以被处理，就好像它实际存在一样，它允许用户生成逻辑层次而无需创建物理实体。

13.3.6 层次序列

由于段是根据层次中它们的序列依次被访问的，所以理解层次是如何排列的很重要。在 IMS 里，段是以自上向下、从左到右的顺序被存储的（图 13-3）。序列是从顶部流向最左边路径的底部。当到达路径的底部后，序列从右边分支路径的顶部继续开始。

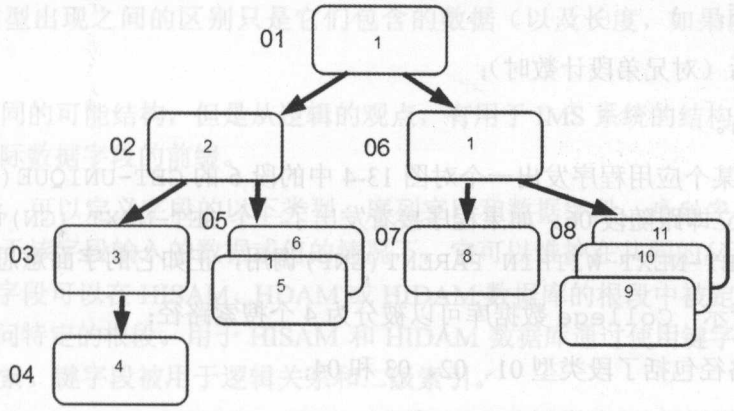


图 13-3 层次中的序列和数据路径

理解记录中段的序列对于理解层次中的移动和位置是很重要的。移动可以向前也可以向后，并且总是沿着层次序列进行。向前移动意味着从上到下，向后移动意味着从下到上。

数据库中的位置意味着在特定段的当前位置。你又做了次深度优先树遍历，但是用了一个稍微不同的术语。

13.3.7 层次数据路径

在图 13-4 中，段里的数字展示了搜索路径遵循的层次结构。每个段左边的数字表示段类型，它们根据类型而不是出现的次数编号。那就是说，04 段类型可以出现任意次数，但 04 *只代表一种类型。段的类型与段编码相关。*

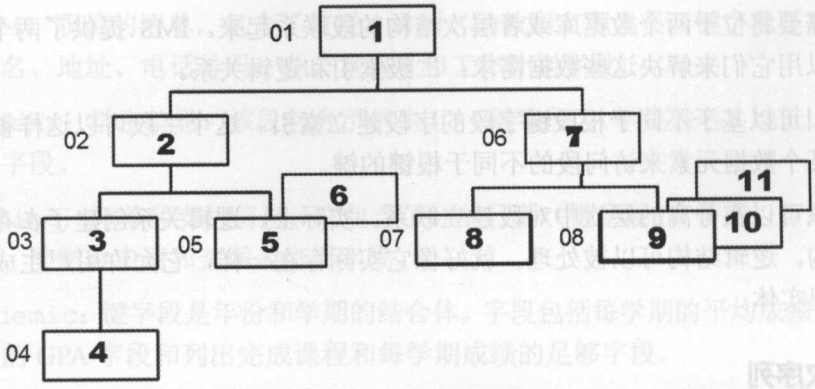


图 13-4　层次数据路径

为了检索某个段、计算路径中每个段类型出现的次数和沿着层次结构前进，可以根据以下规则进行导航：

- 从上到下；
- 从前到后（对兄弟段计数时）；
- 从左到右。

例如，如果某个应用程序发出一个对图 13-4 中的段 6 的 GET-UNIQUE(GU)调用，层次中的当前位置会立即跟随段 06。如果程序接着发出了一个 GET-NEXT (GN)调用，IMS 会返回段 07。还有 GET-NEXT WITHIN PARENT(GNP)调用，正如它的字面意思一样。

如图 13-4 所示，College 数据库可以被分为 4 个搜索路径：

- 第一条路径包括了段类型 01、02、03 和 04；
- 第二条路径包括了段类型 01、02 和 05；
- 第三条路径包括了段类型 01、06 和 07；
- 第四条路径包括了段类型 01、06 和 08。搜索路径总是从根段 01 开始。

13.3.8　数据库记录

一个数据库由一条或多条数据库记录组成，数据库记录由一个段或多个段组成。在 College 数据库中，一条记录由根段 College 和它的依赖段组成，也可以只用根段来定义数据库记录。数据库只能包含定义的记录结构，数据库记录只能包含定义的段类型。

记录这个术语也可用来指数据集记录（或者块），它和数据库记录不是一样的。在数据集中，IMS 使用标准数据系统管理方法存储它的数据库。数据集的最小实体也称为记录（或

者块)。

这两个区别是很重要的。数据库记录可能被存储于一些数据集块中。一个块可能包含几条完整的记录或者几条记录的一部分。在本章，我试着区分数据库记录和数据集记录，它们的含义可能会引起歧义。

13.3.9 段格式

段是数据库中最小的结构，就这个意义来说，IMS 不能检索小于段的数据量。段能被拆分为更小的增量，称为字段，它能被应用程序单独定位。

一条数据库记录最多能包含 255 个段类型。任意类型段出现的数量只受限于你分配给数据库的空间。段类型可以为固定长度或可变长度。你必须定义每个段类型的大小。

区别段类型和段出现的不同是很重要的。`Course` 是为 `College` 数据库在 DBD 中定义的段类型。`Course` 段类型可以出现任意次。每次出现的 `Course` 段类型在 DBD 中都是定义的。段类型出现之间的区别只是它们包含的数据（以及长度，如果段被定义为可变长度)。

段有几个不同的可能结构，但是从逻辑的观点，有用于 IMS 系统的结构和控制信息的前缀，并且 3 是实际数据字段的前缀。

在数据部分，可以定义字段的以下类型：序列字段和数据字段。*序列字段*经常作为*键字段*。它可以在基于该字段输入的数据或值的情况下，它可以维护在共同的父母下出现的序列中的段类型。键字段可以在 HISAM、HDAM 或 HIDAM 数据库的根段中被定义，从而可以让应用程序直接访问特定的根段。用于 HISAM 和 HIDAM 数据库通过使用键字段可以使得数据库记录被顺序检索。键字段被用于逻辑关系和二级索引。

键字段不仅能包含数据还能以特殊的方式帮助用户组织自己的数据库。使用键字段，用户可以用某种键序列来维护某个段类型数据的出现情况，就看怎么设计了。例如，在样例数据库里用户可能想要以学生 ID 升序的方式来保存学生记录。要想这么做，就需要定义学生 ID 为唯一键字段。IMS 以数字的升序方式存储记录。还能通过定义名称字段为唯一键的方式将记录以字母的顺序进行排序。键字段需要记住的 3 个因素如下：

(1) 键字段的数据和值称为段的键；

(2) 键字段可以被定义为唯一或非唯一；

(3) 不必在每个段类型里定义键字段。

通过定义*数据字段*来包含存储在数据库中的实际数据。(记住序列字段就是数据字段。)数据字段，包含序列字段，可以被应用程序定义为供 IMS 使用。

13.3.10　段定义

在 IMS 中，段通过它们出现的顺序以及与其他段的关系被定义。

- 根段：记录中第一，或最高的段。每条记录只能有一个根段。数据库可以有很多条记录。
- 依赖段：数据库中除了根段以外的所有段。
- 父段：在层次结构中，在其之下有一个或多个依赖段的段。
- 子段：在层次结构中，依赖于自己之上的另一个段的段。
- 兄弟段：单个父段下，与同类型的一个或多个段共存的段。

还有一些编辑、加密或压缩段，这里不作考虑。重点是在 IMS 的物理层面上有很多控制数据的方法。

13.4　小结

“那些不能记住过去的人注定要重蹈覆辙。”

——George Santayana

在 SQL 之前有很多数据库，它们都基于导航模型。SQL 程序员不太喜欢承认并不是所有的商业信息都存在于 SQL 数据库中。大多数的商业信息仍存在于简单文件或者旧的、导航型的、非关系型的数据库中。

即使是基于自己特性的新工具已经被接受，成为独立的种类，旧系统的思维模式仍然挥之不去。旧模式依然在新技术中重复。

甚至是早期的 SQL 产品也掉进了这个圈套。例如，如今很多 SQL 程序员使用 IDENTITY 或其他自增型的厂商扩展作为 SQL 表的键，而并不知道它们其实是在模仿 IMS 的导航型序列字段（又名“键字段”）。

这并不是说层次数据库不是组织数据的好方法，它是的！但是你需要知道除去任何具体实现后的抽象含义。SQL 是申明式语言，而 DL/I 是宿主语言里的一系列过程调用。继续用你在 COBOL、PL/I 或任何你使用过的宿主语言里编写过程式代码的风格编写 SQL 代码是一种诱惑。

坏消息是可以使用游标来模仿顺序文件的例程。大体上，READ() 命令变成了内嵌的 FETCH 语句，OPEN 和 CLOSE 文件命令映射到了 OPEN CURSOR 和 CLOSE CURSOR 语句，并且每个文件都变成了没有任何限制并且是某种“记录编号”的简单表。通过这种映射，遗留代码的转换几乎是毫不费力的。并且这也是使用 SQL 数据库编程最坏的形式。

幸好，本书将为你展现一些技巧，让你用 SQL 的方式编写 SQL，而不是在里面夹杂以前语言的方式。

总结思考

IMS 差不多有 50 年的历史，已经从早期的 IBM System/360 技术和 COBOL 转移到了 LINUX 和 Java。不理解所有的数据是如何被建模和被访问的就意味着无法得到那些数据。我个人曾经由于不知道目前流行的应用编程语言而被嘲笑。当我问他们在做什么时，他们正在将 IBM 3270 终端应用移植到手机上。它的后端是个 IMS 数据库，从 1970 年就部署在他们的保险公司了。就像 Alphonse Karr 在 1839 年说的："Plus ça change, plus c'est la même chose" —— "以不变应万变"。

参考文献

www.dbazine.com 网站有一个详细的关于 IMS 的 3 部分的教程，本章的内容就是从这个教程中提炼和总结的。

IMS 资料的最佳来源是 http://www.redbooks.ibm.com/，在这里可以直接从 IBM 下载手册。

Dutta, A. (2010, 2012). *IMS concepts and database administration*. Houston, TX. https://communities.bmc. com/docs/DOC-9908.

Meltz, D., et al. (2005). *An introduction to IMS: Your complete guide to IBM's information management system*. Upper Saddle River, NJ: IBM Press (Pearson Education).

术 语 表

ACID：原子性、一致性、隔离性和持久性，是经典事务处理系统的 4 个理想属性。每个事务的成功或失败都是以原子（不可分割的）单位来衡量的。从事务开始到结束该数据库始终保持着一致的状态，没有违反任何约束。对于数据库，每个事务和其他事务都是隔离开的。最后，事务成功执行后，每个事务完成的工作都将保存（持久化）在数据库中。见 BASE。

数组（array）：一种数据结构，使用一个或多个数值位置索引（下标）来定位值。数组元素的数据类型相同。

BASE：取 basically available（基本可用）、soft state（软状态）和 eventual consistent（最终一致性）的首字母缩写而成。这是“no SQL”版的 RDBMS 的 ACID 属性。假设一个系统能“追上”事实，那么我们是可以接受它是不完整和不准确的。我们愿意等待以获取数据（基本可用），可以接受不同用户看到的相同数据稍微不一致的版本（软状态），并且相信数据最终都会达到一种一致的状态。然而代价是数据的准确性。

批处理（batch processing）：是将一大批事务作为单个工作单位提交的技术。通常情况下，这就是大量数据是如何在数据库中被插入，更新或删除的方式。

Bertillon 卡（Bertillon card）：来自于法国，基于人体测量值复杂系统的早期生物特征识别系统，曾被用作识别罪犯。现在，已经被指纹、DNA 和其他生物识别标识所取代。

大数据（Big Data）：当下热门词汇，通常和 NoSQL 以及混合数据源联系在一起。在 Forrester Research 的文献中，他们用一个朗朗上口的短语 4V 来定义大数据。第一个 V 代表规模，它以指数级增长。第二个 V 代表速度，由于光纤网络等通信系统的改善，数据到达速度比以前更快了。第三个 V 代表多样性，由于设备及其应用程序的多样性，数据源也有所增加。

例如，以前并不存在社交网络、博客和网络摄像头，而现在不一样了。第四个 V 代表可变性，数据格式的种类在不断增加，而这种增加不仅仅是针对关系型或者传统数据存储。其他人还试着对大数据的定义赋予更多的“V”。例如校验、价值、准确性和邻近性。

生物特征识别（biometrics）：是一种基于人类身体特征的测量方式，如针对指纹、视网膜等的识别。见 Bertillon 卡。

Brewer 定理（Brewer's theorem）：以 Eric Brewer 2000 年在 Principles of Distributed Computing（PODC）的探讨会上提出的理论来命名。见 CAP。

字节（byte）：由 8 位二进制数字组成的计算机存储单位。这个术语是在 IBM Stretch 计算机的早期设计阶段，由 Werner Buchholz 在 1956 年 7 月提出的。使用 1 024 位=1 千字节（即 1 KB）这种基本单位可以有几种方法来表示计算机存储的大小，不断复制十进制幂的 SI 前缀来作为基本单位的幂。

1 024	KB	kilobytes
$1\,024^2$	MB	megabytes
$1\,024^3$	GB	gigabytes
$1\,024^4$	TB	terabytes
$1\,024^5$	PB	petabytes
$1\,024^6$	EB	exabytes
$1\,024^7$	ZB	zettabytes
$1\,024^8$	YB	yottabytes

到 2013 年为止，已经有几个商业数据库都是用 petabytes 为单位来衡量的，如沃尔玛的数据仓库。Oracle 曾在他们的数据仓库产品广告中使用前缀“exa”，但在安装中从没实际使用过这一单位。显然将前缀“yotta”用于广告中会显得愚蠢可笑。

CAP：这是由 Eric Brewer 提出的设想。代表了一致性、可用性，以及作为分布式数据库的事务模型的分区容错性。CAP 定理指出，一台分布式计算机系统是不可能同时具备以下 3 个系统特征的：

- 一致性（同一时间点，在分布式数据库中所有节点看到同样的数据）；
- 可用性（每一个数据库请求都能收到一个关于它成功或失败的回应）；
- 分区容错性（无论是信息丢失还是系统部件故障，系统都要继续运行）。

2002 年，Seth Gilbert 和麻省理工学院的 Nancy Lynch 正式发布了对 Brewer 猜想的校验。

见 Brewer 定理。

云计算（cloud computing）：使用互联网的远程服务器进行存储、管理和处理数据的方法，而不是使用本地服务器。云计算一词主要被用为任何通过网络进行分布式计算。

CODASYL：定义 COBOL 的委员会提出了导航数据库模型标准。他们关于数据库的工作被 ANSI X3H2 数据库标准协会接管了。该协会将 CODASYL 进行规范，并制定了 NDL（网络数据库语言）规范。但 SQL 标准一发布，NDL 标准就因缺乏关注而淡出了。

Codd：Edgar Frank“Ted”Codd（2003 年逝世）发明了关系型数据库管理模型，该模型是关系型数据库的理论基础。他的数学方法影响了后来的所有数据库模型。尤其是，他发表了 RDBMS 的 12 条规则，来定义关系型数据库的构成。后来 Codd 博士又提出了联机分析处理（OLAP）这个词，并写了 OLAP 的 12 条定律。

列式数据库（columnar databases）：一种关系型数据库，通常使用 SQL 将数据以列的方式而不是以行的方式存储。其中一个优势是每列数据有且只有一种数据类型。因此，这些数据可以被压缩，还可以用这些列式存储创建很多不同的表。

`COMMIT` 语句（`COMMIT` statement）：是将用户会话中的数据持久化在数据库中的语句。它最初用于具备 ACID 模型的数据库，现在适用于任何数据库。在 ACID 模型中，这些持久化的数据必须满足所有约束条件并保持一致性；在 BASE 模型中，数据仅仅是持久化在数据库中，而我们需要等待来发现它是否一致。见 `ROLLBACK` 语句和 `SAVEPOINT` 语句。

复杂事件处理（complex event processing，CEP）：这意味着并不是所有的数据一开始就已经到达数据库！目前你手中仅有部分数据，期待得到更多的数据来完成查询。这些数据可以来自同一数据源也可以是多个数据源。事件模型和系统状态转换模型不太一样，状态转换是完整性检查，保证了数据只根据流程序列、固定或者可变时间区间序列变化。

压缩（compression）：以更紧凑的形式存储数据，从而可以节省磁盘空间，并且加快数据从次存储器到主存储器的传输速度。一些类型的数据被压缩后，会导致部分原始信息丢失。如音乐或图形，但也可以是无损的。这意味着我们可以从压缩数据来完整重建原始数据。

连通图（connected graph）：根据图论，一系列节点中的任意两个节点都可以通过一个通路到达。这对于查询来说是一个很好的属性，因为任意两个节点可以被关联起来（尽管两个节点之间的关系可能极其遥远）。

一致性：在数据库模式中，只持久化满足模式中的所有约束条件（无论是显式的还是隐式的）的数据。

CQL（Contextual Query Language）：上下文查询语言。由 ANSI Z39 定义的文本数据库的基于正则表达式的字符串模式搜索语言。它类似于如 LexisNexus 和 Westlaw 这样的商业文

本搜索语言。见正则表达式和文本数据库。

环路（cycle）：回到开始节点的路径图，在图论中称为回路。哈密顿回路是一个包含所有节点的图。在 RDBMS 中，它是指关联行为之间的循环引用。这样数据检索会陷入一个无休止的循环中，所以这是不可取的。

Cypher：一种仍在发展和成长中的声明式图查询语言，它使 SQL 程序员的工作更轻松。

DCL（Data Control Language）：数据控制语言。是控制用户去访问某个模式的 SQL 子语言之一，它不是安全或加密的工具。目前，该术语同样适用于非 SQL 数据库。

DDL（Data Definition Language）：数据定义语言。SQL 子语言之一，用于描述和修改表、视图、索引、过程及其他模式对象。目前，该术语同样适用于非 SQL 数据库。

DML（Data Manipulation Language）：数据操作语言。SQL 子语言之一，用于在模式中实现查询、调用过程和更新数据等任务。它不会改变模式的结构或控制方式。目前，该术语同样适用于非 SQL 数据库。

DNA（deoxyribonucleic acid，脱氧核糖核酸）：这种基因和染色体的化学基础物质，使每个人都是独一无二的。DNA 图谱分析是通过编码高度可变的重复基因序列来进行识别的。这些容易分类的遗传物质单元称为可变数目串联重复序列（VNTR），尤其是短串联重复序列（STRs）。DNA 图谱分析是不完整的基因组测序。

文档管理系统（document management systems）：见文本数据库。

边（edge）：图论中称为弧。它们是图结构的一部分，连接两个节点。边可以无向或有向的用来显示关系是对称还是不对称的。它们也可以被赋值，如物理地图建模的图里的距离。

人脸识别（face recognition）：一个生物特征识别的算法集合，可分为两个主要算法：几何学和光度。几何学算法是在人脸上指定一些点，并基于这些点之间的距离和比率创建几何模型，这个网格很容易被编码。光度算法则是尝试将人脸影像与数据库中的基础影像进行匹配。

4V（four V's）：见大数据。

Galton 指纹系统（Galton fingerprint system）：又名 Henry-Galton，是基于指纹外观的指纹分类系统之一。被美国和许多光学匹配系统所使用。

代并发模型（generational concurrency model）：一个并发控制模型，该模型基于在数据库一致时，以数据库在某一时间点（某代）上的状态的“快照”。数据库可被查询或被恢复到快照。这个模型源于为多个用户的相同记录打印多重副本的缩微胶片系统。更新只发生于各种未完成的副本可以被组合成单个一致的更新时。见乐观并发。

地理信息系统（geographic information system，GIS）：专门用于地理或空间数据的数据

库。ISO 技术委员会 211（ISO/TC 211）和开放地理空间联盟（OGC）制定了地理空间标准。OGC 是一个国际行业组织。如今，该组织中的重要成员包括环境系统研究所（ESRI）、计算机辅助资源信息系统（CARIS）、绘图显示和分析系统（MIDAS，现在的 MapInfo）以及地球资源数据分析系统（ERDAS），另外还有两个成立于 20 世纪 70 年代末 80 年代初的公共领域系统（MOSS 和 GRASS GIS）。

图（graph）：一种数学结构，它是由边（又名弧）连接的节点（又名顶点）组成。每条边连接零个节点、一个节点或两个不同的节点。在图表中节点通常被绘制为圆圈或点而边通常被绘制为线。见连通图、边、节点、路径、树和通路。

图数据库（graph database）：用图结构存储关系数据的数据库，它们不太适合于计算和聚合。见边、图、Gremlin、Neo4j 图数据库和节点。

Gremlin：一种开源语言，基于用面向对象和 C 编程语言家族风格的语法对属性图进行的遍历。

散列（hashing）：一种数据访问方式，采用搜索键并对其应用数学函数来得到数据在散列表中的位置。散列表是一个查找表，它保存了在物理存储中数据的实际物理位置。两个不同的键返回相同散列值（碰撞或散列冲突）的情况是有可能发生的，我们必须找到办法来解决这种冲突。一个完美的散列函数是没有冲突的。

HDFS（Hadoop Distributed File System）：Hadoop 分布式文件系统，构建于商用硬件，具有容错性。这个软件的标志是一只可爱的卡通小象。HDFS 和 MongoDB 都是最流行的 NoSQL 数据库版本。

层次数据库系统（hierarchical database system）：基于访问具有层次数据结构的数据的前关系型数据库家族。它是网络或导航型数据库家族的一种特例。见 IMS 和网络数据库系统。

层次三角网格（hierarchical triangular mesh，HTM）：地理定位系统，基于递归的测地线，它用被称作立方像素的三角形来覆盖地球。以 64 位表示，最小的有效 HTM ID 是 8 级但很容易达到 31 级。由于 25 级是在地球表面上约 0.6 m，或 0.02 弧秒，所以对于大部分应用来说，25 级已经足够了。26 级则是在地球表面约 30 cm（小于 1 英尺）。

Hive：这是 Facebook 的一个开源 Hadoop 语言。它比 Pig 更接近 SQL，并且可以像 Pig 一样在不被编译的条件下做特定查询。

信息管理系统（information management system，IMS）：IBM 的产品，是当下最受欢迎的分层数据库。记录中的信息被结构化为记录，记录被细分到关联段的层次树。记录是根段及其所有依赖段，段被进一步细分为多个字段。任何记录中的数据都都关联到一个实体，它稳定、明确、可扩展，而且速度非常快。IMS 数据库通常被为它们设计的访问方法所提及，

如 HDAM（分层直接存取方法）、HIDAM（分层索引直接存取法）、PHDAM（分区 HDAM）、PHIDAM（分区 HIDAM）、HISAM（分层索引顺序访问方法）和 SHISAM（简单 HISAM）。

隔离级别（isolation level）：一种数据库决定会话如何知道其他会话对数据库所做修改的方案的组合。有一种基于提交、未提交工作以及 ACID 属性的 ANSI/ISO 标准 SQL 模型。见作为 ACID 的替代 BASE。

Java：一种编程语言，目前属于 Oracle 公司。JDBC（Java Database Connectivity）是一种基于 Java 的数据访问技术，它定义了客户端如何访问数据库。JDBC 面向关系型数据库，但已然成为面向大多数 NoSQL 产品的最流行的应用程序接口。

键值存储（key–Value store）：具有物理定位符（键）和值的数据存储模型。这些键值对通过键的索引和散列值被搜索并返回值。见 MapReduce。

关键字和题内关键字（KWIC）：是寻找索引或关键字并将其所在位置返回文本数据库中的文本索引技术家族。经典 KWIC 以粗体显示关键字，而关键字左右的文字还是以常规方式显示。该家族包括 KWOC（题外关键字）、KWAC（题内增强关键字）和字母顺序关键词（KEYTALPHA）。这些技术之间的差别在于对用户的显示。

LAMP 栈（LAMP stack）：基于开源软件的网站架构。首字母缩写代表 Linux（操作系统）、Apache（HTTP 服务器）、MySQL（数据库，但是由于它被 Oracle 收购，人们开始转向开源版本 MariaDB），还有 PHP、Perl 或 Python 编程语言。

MapReduce：大型数据集的数据访问技术，该技术依赖于存储的并行使用。首先，`Map()`这一步对数据进行过滤和排序，使数据排成队列。然后，`Reduce()`这一步对从队列中抽取的数据进行汇总和聚合。这个模型是基于在函数式编程语言（如 LISP）中使用的 map 函数和 reduce 函数。

主街道地址指南（Master street address guide，MSAG）：MSAG 描述的是地址元素，包括街道名称和门牌号范围的准确拼写。这是美国邮政服务公司（USPS）的编码准确性支持系统（CASS）的一部分。其他国家也有类似指南。

MDX：微软 OLAP 查询使用的编程语言。由于微软公司在市场上的主导地位，MDX 已经成为事实上的使用标准。

MOLAP：多维联机分析处理。这是电子表格用户首选的“网格数据”版本的 OLAP。见 OLAP 和 ROLAP。

MongoDB：面向文档的开源数据库系统，由 10gen 开发和支持。MongoDB 将结构数据存储为 BSON 格式的文档，BSON 是 JSON（JavaScript 对象表示法）版本之一。它是最流行的 NoSQL 数据库管理系统，有很多第三方工具。

多值数据库（multivalued database）：在文献中这些数据库也称为 NFNF、2NF、NF2 和¬NF。它们不遵循第一范式（1NF），允许表中的某列值是无序列表。这些数据库需要一组运算符对这些结构进行嵌套和解析，另外还要对普通的关系运算符进行适当扩展。

导航数据库（navigational database）：参见网络数据库和 IMS。

Neo4j：最流行的图编程语言。

网络数据库（network database）：一个与 Charles Bachman 相关的前关系型数据库家族。20 世纪 50 年代，Charles Bachman 是陶氏化学的先锋人物。20 世纪 60 年代任职于通用电气，在这里他发明了集成数据存储（IDS），这是第一代数据库管理系统之一。由于对数据库技术的杰出贡献，1973 年 Bachman 获得 ACM 图灵奖。他在获奖演说中就是这样描述该数据库的——由于数据访问的思维模式就像一个读者沿着路径移动从而获取数据，因此网络和分层模型被称为导航数据库。ANSI X3H2 对 NDL（网络数据库语言）制订了一个短暂的从未实施过的标准。

NFNF 数据库（NFNF database）：反第一范式。这些数据库在文献中也被称为 NFNF、2NF、NF2 和¬NF。参见多值数据库。

NIST（National Institute for Science and Technology）：美国国家科学与技术研究院。美国联邦政府的一个机构，它在美国设定了关于 IT 的标准及其他一些标准。18 世纪时，它是以计量与测量局的身份开始。

节点（node）：节点也称为顶点。它是由边连接的图结构中的一部分。节点通常为某种关系的实体或者元素建模。例如，如果我们用图来表示地铁示意图，那么节点就是图中通过轨道连接的地铁站。

NoSQL：一个热门词汇。在文献中被定义为“no sequel”或者（最好）“not only SQL”。通常适用于 MapReduce 数据库，但它有着更普遍（模糊）的含义。

OLAP（online analytical processing）：联机分析处理。这些产品使用在某一时间点时数据库的快照，并对数据提供分析。这就导致了可以通过专门的存储保存结果表或对这些表建立索引的混合 OLAP 能够被复用。如 RDBMS 中的基本表、维度表，以及一些汇总表。这可能是目前这些产品最通用的方法。见 ROLAP 和 MOLAP。

OLTP（online analytical processing）：联机事务处理。它的目的是为日常业务应用程序提供支持。这是在商业市场中 SQL 具有的市场定位。

乐观并发（optimistic concurrency）：数据库的并发控制模型，该模型基于这样的假设：多个用户想在同一时间修改同一数据的概率很小。一旦检测到冲突，回滚到前一个一致的数据库状态。由于之前的数据库状态被保留，它也称为代并发。参见悲观并发。

优化器（optimizer）：数据库引擎的一部分。它尝试从许多可能的等价计划中挑选最好的语句执行计划。与过程式编程不同（告诉机器到底怎么做），这是单向的声明式编程（告诉机器你想要什么）。相同的语句可以有不同的执行计划，因为优化器在调用数据时将使用数据库的当前状态（索引、数据、其他会话中的缓存数据等）。

路径（path）：根据图论，路径是仅经过每个节点一次的通路。如果有 n 个节点，在路径中就有（n−1）边。

悲观并发（pessimistic concurrency）：并发控制模型数据库，该模型基于这样的假设：多个用户总是希望在同一时间修改同一数据。所以记录必须加锁，以防止这种情况的发生。参见乐观并发。

Petri 网（Petri net）：一个数学建模工具，它使用了可以在其节点中保存和移动令牌的图。Petri 网被用于在网络中对并发问题建模。

Pick：本产品一开始是于 1965 年在 IBM System/360 里作为广义信息检索语言系统（GIRLS）被创建的。目前它仍被继续使用并且已成为这种类型的数据库的通用名称。它是一个典型的 NFNF 数据模型，并使用从 BASIC 中的某一个版本发展而来的语言。

Pig Latin：或者直接叫 Pig，是由雅虎开发的查询语言，现在是 Hadoop 项目的一部分。

查询（query）：从数据库中返回结果集而不改变数据的语句。查询并不需要为了备份和恢复而记录日志。然而，对于全面审计，他们需要显示出谁看到什么样的数据以及什么时候看到的。

RAID（redundant array of independent disk）：独立磁盘冗余阵列（原为廉价磁盘冗余阵列）。它是一个磁盘存储硬件阵列家族，基于如果系统有冗余，故障可以通过替换硬件来动态修复而且又不会引起数据库访问故障的概念。

正则表达式（regular expression）：字符串模式匹配系统，最初由数学家 Stephen Cole Kleene 创造。它是 UNIX 的 grep 函数家族和其他编程语言的基础。此工具有许多的供应商和语言版本。ANSI/ISO 标准 SQL 有一个简单的 `LIKE` 谓词和更复杂的 `SIMILAR TO` 谓词。

ROLAP（relational online analytical processing）：关系型联机分析处理。它是在多维 OLAP 之后被开发的。其主要的区别在于 ROLAP 在数据库中并不提前计算或存储汇总数据。

`ROLLBACK` 语句（`ROLLBACK` statement）：一个将数据库恢复到之前的状态并且结束用户会话的语句。它最初用于有 ACID 模型的数据库，但随后应用于不立即持久化改变的任何数据库。参见 `COMMIT` 语句和 `SAVEPOINT` 语句。

`SAVEPOINT` 语句（`SAVEPOINT` statement）：在用户会话中设置保存点的语句，这样一

个事务就可以回滚到保存点被设置时的数据库状态。我们姑且把回滚看成是“几乎提交”。最初用于有 ACID 模型的数据库，但随后应用于不立即持久化改变的任何数据库。参见 COMMIT 语句和 ROLLBACK 语句。

模式/无模式（schema/no schema）：一个对数据库中使用的数据结构的正式描述。在 SQL 中，它是与 DDL（数据定义语言）一起完成的，DDL 描述了表、视图、索引和过程等。其他数据库模型可能有自己的模式语言，或没有一个模式模型。无模式模型通常在相同的存储里内嵌元数据作为数据。最常见的例子是在标记语言和键值对中标签的使用。

SMAQ 栈（SMAQ stack）：发音与“smack stack”相同。这是大数据存储的架构。这些字母分别代表了 storage、MapReduce、assumecommodity hardware with open-source software 以及 query。SMAQ 系统通常是开源的、分布式的，并且在商品硬件上运行。这对于网站来说是平行于商用 LAMP 栈的。LAMP 的字母分别代表市场的 Linux、Apache、MySQL 和 PHP。

Sqoop：一个使用 Java JDBC 数据库 API 的与数据库无关的工具。无论是以批量还是使用查询来限制数据导入的方式，表都能被导入。Sqoop 也具备将 MapReduce 结果从 HDFS 重新注入关系型数据库的能力。

流式数据库（streaming database）：为处理来自数据库控制之外的流数据而设计的数据库。其经典的例子是股票、商品交易以及仪器采样。你可以从 IBM（SPADE）、Oracle（Oracle CEP）、Microsoft（StreamInsight），以及一些类似于 StreamBase（SQL 的面向流的扩展）和 Kx（基于 APL 的 Q 语言）的小供应商以及开源项目（Esper，SQL 的面向流的扩展）找到这类产品。一般来说，所用语言是可读的类 SQL 或是晦涩的类 C。可以看看 StreamBase 和 Kx 这两个极端案例。

文本数据库（textbase）：以前的文档管理系统的现代术语，它们与大量的文本搜索相关。国家信息标准组织 NISO 和 ANSI Z39 小组制定了这个领域的标准。该组织的成员来自于出版、图书馆、互联网技术和媒体机构领域等组织。参见 CQL。

树（tree）：

- 图数据库，这是一个没有回路的连通图。
- 索引，使用许多可能的树结构之一来构建指针链的方式，这些指针链最终会指向某条物理记录。

万国码（Unicode）：为世界上大部分书写系统的一致编码、表示以及文本处理而设立的计算机行业标准。该标准是由 Unicode 协会维护并使用 UTF-8 和 UTF-16 来表示。UTF-8 用 1 个字节表示 ASCII 字符，其在 UTF-8 和 ASCII 编码中具有相同的码值，用 4 个字节表示其他字符。UTF-16 使用两个 16 位单位（4×8 位）来处理每个附加字符。

通路（walk）：来自图论。这是一个连接一组节点且边不重复的边的序列。

WordNet：英语词汇数据库，将英语单词的同义词分组称为同义词集的组。文本数据库用它来做语义搜索。

X3H2：更准确地说是 ANSI X3H2，现在称为 INCITS H2。这是委员会设置的 SQL 语言和其他数据库相关的 IT 标准。ANSI 代表美国国家标准协会，而 INCITS 代表国际信息技术标准委员会（它的发音与 insights 相同）。INCITS 的前身为 X3 和 NCITS。这些年的 SQL 标准有：SQL-86、SQL-89、SQL-9、SQL：1999、**SQL**：2003 和 SQL：2008，以及正在撰写本文时的 SQL：2011。ANSI/ISO 标准每 5 年审查一次，在审查时标准是可被弃用的。

XML（eXtensible Markup Language）：可扩展标记语言。定义了一系列规则，用来以人类可读和机器可读的格式编码文档。这是一种在 Unicode 大力支持下用于世界上各种语言交换数据的国际标准。尽管 XML 的设计专注于文件，但它仍被广泛地用于表示任意数据结构。主要的特性是那些标签，它们被成对地写成<标签>和</标签>的形式将数据括起来。其他特性使用尖括号将它们从数据中分离出来。

邮政编码（ZIP code）：区域改进计划（zone improvement plan，ZIP）码，是 USPS 为地址所使用的地理编码系统。ZIP 码术语已经成为其他国家任何地址位置代码的通用术语。

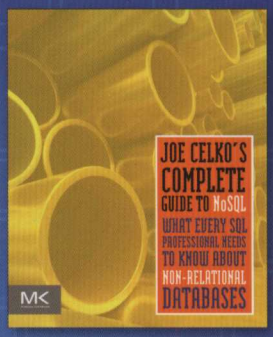
在畅销书作者、SQL 权威人物 Joe Celko 的专业指导下学习 NoSQL。
掌握每一位 SQL 专业人士都需要了解的非关系型数据库知识。

“Joe 再一次既彻底又清晰地阐述了一个很少能被人讲清楚的主题。本书从当前和历史两个方面介绍 NoSQL，展示了 NoSQL 从大型机时代到云时代的发展进程，以及这种发展带来的影响。从 1995 年拿到他的第一本书 *Instant SQL* 开始，他的书一直是我的重要信息来源。”

——Les Cardwell 博士，DCS-DSS，Central Lincoln PUD 企业架构师

“如果你遇到了通过 SQL 模型无法解决的问题，或者你只是想学习数据管理方面的知识，尤其是 NoSQL 方面的，那么 Joe Celko 的书会是你理想的选择。”

——Jeff Garbus，飞鹰咨询合伙人



本书对非关系型技术的完整概述，让你能够更灵活地应对组织需要。数据持续爆炸式增长且越来越复杂，使得 SQL 对于查询数据和抽取数据含义的作用被削弱。在这个新时期里，数据的规模更大、速度更快，你需要利用非关系型技术来最大限度地利用手中的信息。通过 Joe Celko 的这本书，你将了解何时、何地使用 NoSQL 以及为什么 NoSQL 比 SQL 更有益。

- 完整理解 NoSQL 技术的最佳使用场景。
- 认识列式数据库、流式数据库和图数据库的利与弊。
- 学习如何转变思维方式来充分利用非关系型数据库。

本书涵盖了当今的数据不同于过去的数据的三个方面，即速度、规模、多样性。当收集和查询信息的速度赶不上信息变化的速度时，就不能简单地将其当作静态数据处理。本书将帮你理解数据的速度，让你装备一些工具来解燃眉之急。旧的数据存储与访问模式已不适用于大数据。权威专家 Joe Celko 将帮你了解数据的规模，以及存储和访问 PB 级和 EB 级数据的不同方式。并不是所有的数据都能适应关系模型，包括基因数据、语义数据以及社交网络产生的数据。Joe Celko 将帮你了解数据的种类，以及多种数据所需的可替代的存储、查询和管理框架。

异步社区 www.epubit.com.cn
新浪微博 @人邮异步社区
投稿/反馈邮箱 contact@epubit.com.cn

本书译自原版 **Joe Celko's Complete Guide to NoSQL**，并由 **Elsevier** 授权出版。

ELSEVIER
爱思唯尔

ELSEVIER

ISBN 978-7-115-42787-8

ISBN 978-7-115-42787-8
定价：45.00 元

分类建议：计算机/数据库/NoSQL
人民邮电出版社网址：www.ptpress.com.cn